Town Crier:
An Authenticated Data Feed for Smart Contracts

Your N. Here
Your Institution

Second Name
Second Institution

Name
Name Institution

Abstract

Smart contracts are programs that execute autonomously on blockchains. Many of their envisioned uses require them to consume data from outside the blockchain. (For example, a financial instrument might rely on stock prices.) Trustworthy data feeds that can interface with smart contracts will thus be critical to any smart contract system.

We present an authenticated data feed system called Town Crier (TC). TC builds on the observation that many web sites, such as major news and finance sites, already serve as trusted data sources for non-blockchain uses. TC acts as a bridge between such servers and smart contract systems, using trusted hardware to authenticate and scrape data from HTTPS-enabled websites and to generate trustworthy datagrams for relying smart contracts. It also includes a range of advanced features, such as private datagrams, which decrypt and evaluate ciphertext requests within TC's hardware.

We describe the TC architecture, its underlying trust model, and its applications, and report on an implementation that uses the newly released SGX software development kit and furnishes data for the smart-contract system Ethereum. We also present formal proofs of the security of TC, including correct handling of payment in Ethereum. We will soon be launching TC as an online public service.

1 Introduction

2 Background

2.1 SGX

Intel's Software Guard Extensions (SGX) is a set of new instructions that confer hardware protections on user-level code. Its goal is to provide *isolated execution*. SGX enables a process to execute in a protected address space known as an *enclave*. It protects the confidentiality and integrity of a process in an enclave from other software on the same host, including the operating system, as well as from certain forms of hardware attack, such as memory probes.

A enclave process cannot make system calls, nor can it execute code outside the enclave region. As a means of communicating with processes outside the enclave, however, it can read and write memory outside the enclave region, consistent with OS setting of page permissions. Thus isolated execution in SGX may be viewed in terms of an ideal model in which a process is guaranteed to execute correctly and with perfect confidentiality, but relies on a (potentially malicious) operating system for network and file-system access.[1]

Another feature of SGX is its support for *attestation*, which allows a remote system to verify the software in an enclave and communicate securely with it. When an enclave is created, the CPU produces a hash of its initial state known as a *measurement*. The software in the enclave may at a later time request a report, which includes a measurement and any supplementary data provided by the process, such as a public key and timestamp. The report may be digitally signed (by a trusted process called a "quoting enclave") using a hardware-protected key to produce a proof that the measured software is running in an SGX-protected enclave. This proof, known as a *quote*, may be verified by a remote system, while the associated public key in the supplementary data can then be used by the remote system to establish a secure channel with the enclave or verify signed data it emits. We use the generic term *attestation* to refer to a quote, and denote it by `att`. We assume that a trustworthy measurement of the code for the enclave component of TC is available to

[1] This model is a simplification, as SGX is known to expose some of the internal state of an enclave to the operating system (e.g., making page faults visible to the exception handler []), creating potential side-channel vulnerabilities. We regard side channels as outside the scope of this paper, and thus assume an ideal model of isolated execution.

any client that wishes to verify an attestation. SGX signs quotes using a *group signature* scheme called EPID []. This choice of primitive is significant in our design of Town Crier, as EPID is a proprietary signature scheme and is not supported in Ethereum.

SGX additionally provides a trusted time source via the function `sgx_get_trusted_time`. On invoking this function, an enclave obtains a measure of time relative to a reference point labeled with a nonce. We refer to this as the *clock reference point*. It remains stable for a given nonce, but SGX does not provide a source of absolute or wall-clock time, a limitation that we must work around in TC.

2.2 HTTPS

2.3 Smart contracts

Smart contracts are the expression of contractual agreements, including financial instruments, as executable code. In the context of cryptocurrencies, the term refers specifically to autonomously executing scripts that reside on a blockchain and can manipulate control currency. Bitcoin has a scripting language that can serve to implement a limited form of smart contract, but it is not Turing-complete and lacks support for loops.

Ethereum is the first decentralized blockchain with a Turing-complete scripting language and thus full support for smart contracts. Other Turing-complete smart contract systems exist, such as Counterparty [], which runs as a Bitcoin overlay, but is not fully decentralized. Ethereum has its own associated cryptocurrency called *ether*. (At the time of writing, 1 ether has a market value of a little more than $2 U.S.) While TC can be adapted in principle to any smart contract system, we report on an implementation directed at Ethereum.

A smart contract in Ethereum is represented as what is called a *contract account*, endowed with code, a currency balance, and persistent memory in the form of a key/value store. Contract code executes in response to receipt of a *message* from another contract or a *transaction* from a non-contract (*externally owned*) account, informally what we call a "wallet." Thus, contract execution is always initiated by a transaction. Informally, a contract only executes when "poked," and poking progresses through a sequence of entry points until no further message passing occurs (or until there is a shortfall in gas, as explained below).

A smart contract accepts messages as inputs to any of a number of designated functions. These entry points are determined by the contract creator and represent the API of the contract. Once created, a contract executes autonomously; it persists indefinitely, with even its creator unable to modify its code. (There's one exception: a special opcode `suicide` will wipe code from a contract account.) As a simple abstraction, then, a smart contract may be viewed as an *autonomous agent* on the blockchain.

To prevent denial-of-service attacks or inadvertent infinite looping within contracts and in general to control resource expenditure by the network, Ethereum implements uses a resource called *gas* to power contracts. Opcodes in smart contracts have globally specified, fixed gas costs, as do the use of a contract's persistent storage and the data in transactions and messages. Transactions and messages include a parameter (`STARTGAS`) specifying a bound on amount of gas expended by the computations they initiate. Gas is carried along the execution path induced when a transaction or message is passed to a contract, and depleted as instructions are executed or reads or writes are made to persistent storage. Should a function fail to complete due to a shortfall in gas, it is aborted and any state changes induced by the partial computation are rolled back to their pre-call state; previous computations along the call path, however, are retained.

Along with the `STARTGAS` parameter, a `GASPRICE` parameter is included that specifies the maximum amount in ether that the transaction is willing to pay per unit of gas. The transaction thus succeeds only if the initiating account has a balance of `STARTGAS` $\times$ `GASPRICE` ether and `GASPRICE` is high enough to be accepted by the system (miner).

The management of gas, as we show in our design of Town Crier can be delicate. Without careful construction, for example, the smart contracts representing TC's interface on the Ethereum blockchain can be caused by an attacker to exhaust the ether used to power the delivery of datagrams.

2.4 Applications of ADFs for smart contracts

2.5 Basic terminology

We refer to a *smart contract* making use of the Town Crier service as a *relying contract*. In contexts where a relying contract has issued to TC a service request for a datagram, we call it a *requester*. We denote a requester by $\mathcal{C}_U$. A party (or would-be party) to a relying contract, a person, organization, or server, is a *client*. Relying contracts—and requesters, by extension—are blockchain entities, while a client is an off-chain entity. A *data source*, or *source* for short, is an online server (running HTTPS) that provides data which TC uses to compose datagrams.

3 TC Architecture and Security Model

The Town Crier system includes three main components: The TC Contract ($\mathcal{C}_{TC}$), the Enclave ($\mathcal{E}$), and the Relay ($\mathcal{R}$). The Enclave and Relay reside on the TC server, while the TC Contract resides on the blockchain. An architectural schematic showing the roles of these components is given in Figure 1.

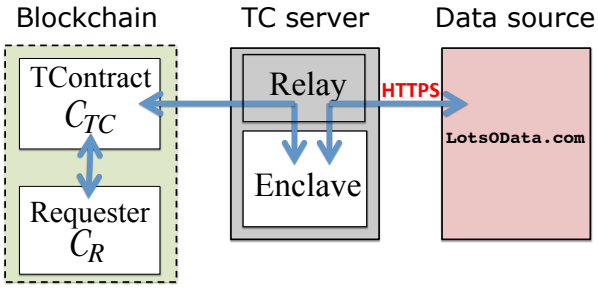

Figure 1: **Basic Town Crier architecture.**

The TC Contract $\mathcal{C}_{TC}$. The TC Contract (denoted by $\mathcal{C}_{TC}$) is a smart contract that acts as the blockchain front end of the TC service. It is designed to present a simple API to a relying contract $\mathcal{C}_U$ for its requests from TC. Very simply, $\mathcal{C}_{TC}$ accepts datagram requests from a requester $\mathcal{C}_U$ and returns corresponding datagrams from TC. Additionally, $\mathcal{C}_{TC}$ manages TC monetary resources, which in Ethereum take the form of ether (money) and gas ("fuel" for contracts).

The Enclave $\mathcal{E}$. We refer to the TC code running in the SGX enclave simply as the Enclave. In TC, the Enclave ingests and fulfills datagram requests from the blockchain. To obtain the data for inclusion in datagrams, it queries external data sources, specifically HTTPS-enabled internet services. It returns a datagram to a requesting contract $\mathcal{C}_U$ as a digitally signed blockchain message. Under our assumed security model for SGX, apart from its network functions, the Enclave runs in complete isolation from an adversarial OS as well as other process on the host.

The Relay $\mathcal{R}$. As an SGX enclave process, the Enclave lacks direct network access. Thus the Relay handles bidirectional network traffic on behalf of the Enclave. Specifically, the Relay provides network connectivity from the Enclave to three different types of entities:

1. *The Blockchain (the Ethereum system):* The Relay scrapes the blockchain in order to monitor the state of the TC Contract $\mathcal{C}_{TC}$. In this way, it performs implicit message passing from $\mathcal{C}_{TC}$ to the Enclave, as neither component itself has network connectivity. Additionally, the Relay places messages emitted from the Enclave (datagrams) on the blockchain.

2. *Clients:* The Relay runs a web server to handle off-chain service requests from clients, specifically, requests for attestations from the Enclave. As we soon explain, an attestation provides a unique public key for the Enclave instance to the client and proves that the Enclave is executing correct code in an enclave and that its clock is correct in terms of absolute (wall-clock time). A client that successfully verifies an attestation can then safely create a relying contract $\mathcal{C}_U$ that uses the TC.

3. *Data sources:* The Relay relays traffic to and from data sources (HTTPS-enabled servers) queried by the Enclave.

The Relay is an ordinary user-space application. It does not benefit from integrity protection by SGX and thus, unlike the Enclave, can be subverted by an adversarial OS on the TC server, causing network delays or failures. A key design aim of TC, however, is that Relay should be unable to cause incorrect datagrams to be produced or users to lose fees paid to TC for datagrams (although they may lose gas used to fuel their requests). As we shall show, in general the Relay *can only mount denial-of-service attacks against TC*.

Security model Here we give a brief overview of our security model for TC, providing more details in later sections. We assume the following:

- *Blockchain communication.* Transaction and message sources are authenticable, i.e., a transaction m sent from an account $\mathcal{P}_X$ (or message m from contract $\mathcal{C}_X$) is identified by the receiving account as originating from X. Transactions and messages are integrity protected (as they are digitally signed by the sender), but not confidential.

- *Enclave security:* We make three assumptions about Enclave: (1) The Enclave behaves honestly, i.e., correctly executes the TC protocol; (2) The private key $\mathsf{sk}_{\mathrm{TC}}$ is known only the Enclave; and (3) The Enclave has an accurate (internal) real-time clock. (Specifically, the clock is accurate to within [Ari: XXX], as we show experimentally.) We explain in the next section how we achieve these properties through use of SGX and how the public key $\mathsf{pk}_{\mathrm{TC}}$ may be bound to an Ethereum account, given the Enclave an authenticable blockchain presence.

- *Network communication.* The Relay (and other untrusted components of the TC server) can tamper

with or delay communications to and from the Enclave. (As we explain in our SGX security model, the Relay cannot otherwise observe or alter the behavior of the Enclave.) Thus the Relay is subsumed by an adversary that controls the network.

4 TC Protocol

We now describe the operation of TC at the protocol level. The basic protocol is conceptually simple: A user contract $\mathcal{C}_U$ requests a datagram from the TC Contract $\mathcal{C}_{TC}$. $\mathcal{C}_{TC}$ forwards the request to $\mathcal{E}$ and then returns the request to $\mathcal{C}_U$. There are many details, however, relating to message contents and protection and the need to connect the off-chain parts of TC with the blockchain.

First, we give a brief protocol overview. Then we enumerate the data flows in TC. Finally, we provide a component-level view of the protocol by specifying the functionalities embodied in the TC Contract, Relay, and Enclave. We present these as ideal functionalities, inspired by the universal-composability (UC) framework, in order to abstract away implementation details and as a springboard for formal proofs of security. We omit details in this section on how payment is incorporated into TC; this delicate aspect of the system design is deferred to Section **??**.

4.1 Datagram lifecycle

The lifecycle of a datagram may be briefly summarized in the following steps:

- **Initiate request.** $\mathcal{C}_U$ sends a datagram request to $\mathcal{C}_{TC}$ on the blockchain.
- **Monitor and relay.** The Relay monitors $\mathcal{C}_{TC}$ and relays any incoming datagram request with parameters params to the Enclave.
- **Securely fetch feed.** To process the request specified in params, the Enclave contacts a data source via HTTPS and obtains the requested datagram. It forwards the datagram via the Relay to $\mathcal{C}_{TC}$.
- **Return datagram.** $\mathcal{C}_{TC}$ returns the datagram to $\mathcal{C}_U$.

We now make this data flow more precise.

4.2 Data flows

A datagram request by $\mathcal{C}_U$ takes the form of a message $m_1 = (\mathsf{id}, \mathsf{callback}, \mathsf{params})$ to $\mathcal{C}_{TC}$ on the blockchain. Here, id is a unique request identifier (which we explain later how to compute in practice); callback specifies the entry point in $\mathcal{C}_U$ to which the datagram is to be returned. (In principle, callback could specify an entry point in a different contract, but TC does not yet adopt this generalization. params specifies the requested datagram, e.g., $\mathsf{params} := (\mathsf{url}, \mathsf{spec}, T)$, where url is the target data source and spec specifies content of a the datagram to be retrieved (e.g., a stock ticker at a particular time), while T specifies the delivery time for the datagram.

$\mathcal{C}_{TC}$ forwards $m_2 = (\mathsf{id}, \mathsf{params})$ to the Enclave. It receives in return a return message $m_3 = (\mathsf{id}, \mathsf{params}, \mathsf{data})$ from the TC service, where data is the datagram, i.e., contains the data (e.g., the desired stock ticker price). $\mathcal{C}_{TC}$ checks the consistency of params on the incoming and outgoing messages, and if they match forwards data to the entry point callback in $\mathcal{C}_U$ in message m_4.

Fig. 2 shows the data flows involved in processing a datagram request. For simplicity, the figure omits the Relay, which is only responsible for data passing.

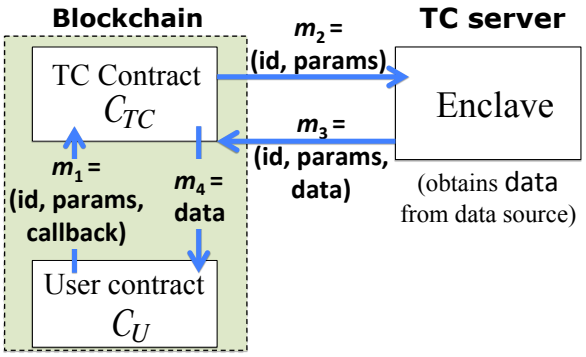

Figure 2: **Data flows in datagram processing.**

Digital signatures are needed to authenticated messages, such as ans, entering the blockchain from an external source. We let $(\mathsf{sk}_{\mathrm{TC}}, \mathsf{pk}_{\mathrm{TC}})$ denote the private / public keypair associated with the Enclave for such message authentication. For simplicity, Fig. 2 assumes that the Engine can send signed messages directly to $\mathcal{C}_{TC}$. We explain shortly how Ethereum requires a slightly different approach in which TC sends messages via an Ethereum wallet $\mathcal{P}_{TC}$.

4.3 Use of SGX.

Our protocols in TC rely on the ability of SGX attestation to bind an instance of the Enclave to a public key and for the Enclave to maintain accurate absolute or wall-clock time. To simplify our presentation, we defer most details of our formal model of SGX capabilities and specification of protocols for attestation generation to the paper appendix. Instead, we first explain and present a simple abstraction of these capabilities suitable for our TC protocol descriptions.

Let $\mathsf{prog}_{\text{encl}}$ represent the code for Enclave, which we presume is trusted by all system participants. To avoid having to verify an EPID group signature on the blockchain, we have clients obtain SGX attestations from the Relay and verify them off-chain. As noted above, it is possible to bind an enclave process instance running $\mathsf{prog}_{\text{encl}}$ to a key pair $(\mathsf{pk}_{\text{TC}}, \mathsf{sk}_{\text{TC}})$ by including the public key pair in an attestation. We take this approach in TC.

Binding $\mathsf{prog}_{\text{encl}}$ to Ethereum wallet $\mathcal{P}_{TC}$. Information can only be inserted into the blockchain in Ethereum as a transaction from a wallet. Thus, the only way the Relay can relay messages form the Enclave to $\mathcal{C}_{TC}$ is through a wallet (again, formally, an externally owned account) $\mathcal{P}_{TC}$. Since the Relay may corrupt messages, however, it is critical that they be authenticated by the Enclave. Since Ethereum itself already verifies signatures on transactions from externally owned accounts (i.e., users interact with the Ethereum blockchain through an authenticated channel), TC uses a trick to *piggyback verification of enclave signatures on top of Ethereum's already existing transaction signature verification mechanism*.

Very simply, the Enclave creates $\mathcal{P}_{TC}$ with the public key pk_{TC}.

To make this idea work fully, the public key pk_{TC} must be hardcoded into $\mathcal{C}_{TC}$. A client creating or relying on a contract that uses $\mathcal{C}_{TC}$ is responsible for making sure that this hardcoded pk_{TC} has an appropriate SGX attestation before interacting with the $\mathcal{C}_{TC}$ blockchain contract. Let Verify denote a verification algorithm for EPID signatures. Fig. 3 gives the protocol for a client to check that $\mathcal{C}_{TC}$ is backed by a valid Enclave instance. This protocol does not include a mechanism for *revocation* of a compromised SGX instance, an issue we discuss later in the paper.

In summary, then, we may assume in our protocol specifications that *all relying clients have verified an attestation for Enclave and thus that datagram responses passed from $\mathcal{P}_{TC}$ to $\mathcal{C}_{TC}$ are trusted to originate from $\mathcal{E}$.*

Clock. Additionally, as noted above, trusted clock provides only relative time with respect to a reference point, not absolute time. Thus, when initialized, the Enclave is provided with the current wall-clock time by a trusted source, e.g., the Relay (under a trust-on-first-use model). The SGX attestation generated by the Enclave includes the current wall-clock time, which clients may verify in real time. Thus, a client can determine the absolute clock time of Enclave to within a high degree of accuracy, bounded by the round-trip time of its attestation request plus the attestation verification time–on the order of hundreds of milliseconds in a wide-area network []. A high degree of accuracy is potentially useful for some applications but sub-second accuracy is not required for most. Ethereum has a block interval of 12s and the clock serves in TC primarily to: (1) Schedule connections to data sources and (2) To check TLS certificates for expiration when establishing HTTPS connections. [Ari: Let's give the attestation API a function name.] We present a formal model and protocol specification in the paper appendix.

Notation. We model execution in SGX in terms of a functionality $\mathcal{F}_{\text{sgx}}$ operating in a stateful manner on $\mathsf{prog}_{\text{encl}}$. This functionality may be be invokes through transmission of one of three messages to $\mathsf{prog}_{\text{encl}}$: `init`, which creates the enclave with $\mathsf{prog}_{\text{encl}}$ as its initial state and triggers measurement quotes, `attest`, which causes $\mathcal{E}$ to initiate an attestation with the public key and current time as supplementary data, and (`resume`, X) which initiates an execution of $\mathsf{prog}_{\text{encl}}$ on a fresh input X. (We assume that $\mathsf{prog}_{\text{encl}}$ exits only when it completes processing of a given input.) We let $\mathcal{F}_{\text{sgx}}[\mathsf{prog}_{\text{encl}}, \mathcal{R}]$ denote invocation of $\mathcal{F}_{\text{sgx}}$ on $\mathsf{prog}_{\text{encl}}$ by $\mathcal{R}$. We let $\mathsf{clock}()$ denote measurement of the SGX clock from within the enclave; $\mathsf{clock}()$ returns the current wall-clock time (in a canonical format such as seconds since the Unix epoch January 1, 1970 00:00 UTC).[Ari: What format do we actually use?]

We abstract away the use of group signatures in EPID and simply denote the keypair associated with an SGX instance in TC by $(\mathsf{pk}_{\text{sgx}}, \mathsf{sk}_{\text{sgx}})$. We let $\Sigma.\mathsf{Sign}(\mathsf{sk}, X)$ denote a digital signature under private key sk of message X, and $\Sigma.\mathsf{Verify}(\mathsf{sk}, \sigma, X)$ denote the corresponding verification operation.

User: offline attestation of SGX enclave

Inputs: pk_{sgx}, pk_{TC}, $\mathsf{prog}_{\text{encl}}$, σ_{att}

Checks:
Assert $\mathsf{prog}_{\text{encl}}$ is the expected enclave code
Assert $\Sigma_{\text{sgx}}.\mathsf{Verify}(\mathsf{pk}_{\text{sgx}}, \sigma_{\text{att}}, (\mathsf{prog}_{\text{encl}}, \mathsf{pk}_{\text{TC}}))$
Assert $\mathcal{C}_{TC}$ is correct and parametrized w/ pk_{TC}
// now okay to rely on $\mathcal{C}_{TC}$

Figure 3: A client checks an SGX attestation on the enclave's code $\mathsf{prog}_{\text{encl}}$ and public key pk_{TC}; the client checks that pk_{TC} is hardcoded into TC blockchain contract $\mathcal{C}_{TC}$ before using $\mathcal{C}_{TC}$.

4.4 A payment-free basic protocol

For simplicity, we first specify functionalities in a payment-free version of our basic protocol, i.e., one that

does not include gas or fees. Later, in our implementation discussion, we explain how we handle these two forms of payment, and we prove payment-related properties in the paper appendix. For simplicity, we assume a single instance of $\mathcal{E}$ and a since TC server, although our architecture could scale up to multiple instances of either. To show messages corresponding to those in Fig. 2, we use the label (**msg.** m_i)

The TC Contract $\mathcal{C}_{TC}$. The TC Contract, as noted above, accepts a datagram request $(\mathsf{id}, \mathsf{params}, \mathsf{callback})$, forwards it to the TC server, and sends the resulting datagram data to the entry point callback in the requesting contract $\mathcal{C}_U$. As we explain in Section **??**, the blockchain verifies that the response is correctly signed under $\mathcal{E}$'s key $\mathsf{pk}_{\mathrm{TC}}$, so $\mathcal{C}_{TC}$ need not verify the signature explicitly. TC does, however, have a subtle security requirement. Specifically, for a given datagram request id, $\mathcal{C}_{TC}$ verifies that $\mathsf{params}' = \mathsf{params}$, where params is the digitally signed message produced by $\mathcal{E}$ and params is the locally stored parameters. The check is necessary to prevent $\mathcal{R}$ from corrupting datagram requests passed by $\mathcal{C}_{TC}$ (which, as a public function, has no means of digitally signing requests). $\mathcal{C}_{TC}$ is specified in Fig. 12. As a reminder, we assume (and enforce) the condition that id is unique to a given request.

Program for Town Crier blockchain contract $\mathcal{C}_{TC}$

Request: On recv $(\mathsf{id}, \mathsf{params}, \mathsf{callback})$ from $\mathcal{C}_U$: // **msg.** m_1
Record $(\mathsf{id}, \mathsf{params}, \mathsf{callback})$

Deliver: On recv $(\mathsf{id}, \mathsf{params}, \mathsf{data})$ from $\mathsf{pk}_{\mathrm{TC}}$:
Let $(\mathsf{id}, \mathsf{params}', \mathsf{callback})$ be a most recently recorded tuple corresponding to id :
Assert $\mathsf{params} = \mathsf{params}'$
Call $\mathsf{callback}(\mathsf{data})$ // **msg.** m_4

Figure 4: The Town Crier TC Contract $\mathcal{C}_{TC}$.

The Enclave $\mathcal{E}$. When initialized through an `init` call, $\mathcal{E}$ sets its absolute clock (by setting a reference point) and generates a keypair $(\mathsf{pk}_{\mathrm{TC}}, \mathsf{sk}_{\mathrm{TC}})$ to which it binds by placing $\mathsf{pk}_{\mathrm{TC}}$ in attestation requests. Given input `resume` $(\mathsf{id}, \mathsf{params})$, $\mathcal{E}$ contacts the requested data source via HTTPS and checks that the corresponding certificate cert is valid, i.e., not expired. We defer discussion of certificate revocation to Section **??**. $\mathcal{E}$ then fetches the requested datagram and returns it to $\mathcal{R}$ along with the inputs, all digitally signed. The protocol for $\mathcal{E}$ is shown in Fig. 5.

Program for Town Crier Enclave $\mathcal{E}$ (enclave)

Initialize: On recv (`init`):
$(\mathsf{pk}_{\mathrm{TC}}, \mathsf{sk}_{\mathrm{TC}}) := \Sigma.\mathsf{KeyGen}(1^\lambda)$
Output $\mathsf{pk}_{\mathrm{TC}}$
/*$\mathcal{F}_{sgx}$ *attests to the code and the output* pk_{TC}, see Figure [elaine: refer]*/

Resume: On recv (`resume`, $(\mathsf{id}, \mathsf{params})$)
Parse $\mathsf{params} := (\mathsf{url}, \mathsf{spec}, T)$:
Wait until $\mathsf{clock}() \geq T$
Contact url via HTTPS, obtaining cert
Verify cert is valid for time $\mathsf{clock}()$
Obtain webpage w from url
Parse w to extract data with specification (spec, T)
$\sigma := \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{TC}}, (\mathsf{id}, \mathsf{params}, \mathsf{data}))$
Output $((\mathsf{id}, \mathsf{params}, \mathsf{data}), \sigma)$

Figure 5: The Town Crier Enclave $\mathcal{E}$.

The Relay $\mathcal{R}$. As noted in Section **??**, $\mathcal{R}$ performs distinct three forms of data passing, which we specify more precisely here. It scrapes the blockchain, monitoring $\mathcal{C}_{TC}$ for new datagram requests $(\mathsf{id}, \mathsf{params})$. It boots the $\mathcal{E}$ with an `init` call and handles incoming requests by invoking $\mathcal{E}$ with `attest` or `resume` calls. Finally, it forwards datagram responses from $\mathcal{E}$ to the blockchain; recall that it inserts data onto the blockchain and thus forward responses through $\mathcal{P}_{TC}$. The program for $\mathcal{R}$ is shown in Fig. 13.

Program for Town Crier Relay $\mathcal{R}$

Initialize:
Send `init` to $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$
On recv $(\mathsf{pk}_{\mathrm{TC}}, \sigma_{\mathrm{att}})$ from $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$:
Publish $(\mathsf{pk}_{\mathrm{TC}}, \sigma_{\mathrm{att}})$

Loop forever:
Whenever $\mathcal{C}_{TC}$ receives $(\mathsf{id}, \mathsf{params}, \_)$:
Send (`resume`, $(\mathsf{id}, \mathsf{params})$) to $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$
On recv $((\mathsf{id}, \mathsf{params}, \mathsf{data}), \sigma)$ from $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$:
Send $((\mathsf{id}, \mathsf{params}, \mathsf{data}), \sigma)$ to $\mathcal{C}_{TC}$ // **msg.** m_3

Figure 6: The Town Crier Relay $\mathcal{R}$.

5 Implementation

5.1 TC Blockchain resources

These include the TC contract and the addresses from which it sends messages and manages its wallet

There are two parts to the Town Crier blockchain resources:

- $\mathcal{P}_{SGX}$– An Ethereum wallet whose private key is generated and controlled by the SGX enclave.
- $\mathcal{C}_{TC}$– An Ethereum contract with two entry points, described in Table 12.

The Town Crier server is responsible for identifying calls to $\mathcal{C}_{TC}$'s Request request method by scraping the contract activity on the blockchain. It then replies to those requests through Deliver once it has acquired the necessary data and packaged it into a response.

5.2 Client API

5.3 TC server

5.3.1 Trusted executable

5.3.2 Untrusted executable

6 Experiments

- Total execution time–with PROFILING
- Clock granularity
- Gas costs

7 Security Analysis

7.1 Gas neutrality

Here we assume an adversary which is active on the blockchain, the network, and within the untrusted executable running on the Town Crier server. However, we assume that the adversary will not execute an arbitrary denial of service attack, but will rather delay messages indefinitely and deliver bogus data whenever such data will be accepted as valid. Because operations on the blockchain are verifiable and the SGX enclave can attest to what it is running, we assume those are honest.

In this model we show that, for every request which provides a sufficient fee, a valid authenticated datagram will be delivered to the requested callback location in finite time. If the request includes an insufficient fee (but is otherwise valid), the datagram will not be delivered, but the (too-small) fee will still be collected.

Lemma 1. *If seeded with at least $F_{\max}$ ether, the $\mathcal{P}_{SGX}$ wallet will have at least as much money after each transaction as it had before that transaction.*

Proof. [Ethan: This is actually a proof sketch, I just put it in a proof tag.]

Because all blockchain transactions from $\mathcal{P}_{SGX}$ must be initiated by the SGX enclave and the SGX only calls $\mathcal{C}_{TC}$.Deliver, we need only reason about what happens inside that function. Because transactions including $\mathcal{C}_{TC}$ are transmitted securely into the SGX enclave, it will only see valid requests (ones for which $F_{\min} \leq \$\text{fee} \leq F_{\max}$) and the arguments it sees for those requests will be correct. Moreover, it saves the transaction ID of each request it fulfills and never fulfills a request with the same transaction ID twice. This means that whenever deliver is called, it will be called in connection with a valid request that has not already been delivered. Thus it suffices to show that:

1. The first time a valid request is delivered, $\mathcal{C}_{TC}$ will contain at least \$fee ether.
2. \$fee is never lower than the amount $\mathcal{P}_{SGX}$ must spend in gas.
3. The execution of Deliver will never run out of gas (and thus always succeed).

To prove claim 1 we first note that ether can only be removed from $\mathcal{C}_{TC}$ as part of a call to Deliver from $\mathcal{P}_{SGX}$. Because $\mathcal{P}_{SGX}$ is honest, it will only make this call in connection with a valid request, and the specified value of \$fee will always be the fee submitted with that request. Because $\mathcal{C}_{TC}$ pays out the specified \$fee on the call to Deliver, the ether is always exactly the ether stored from a previous, valid, undelivered request, and thus will always be present in $\mathcal{C}_{TC}$.

To prove claim 2 first note that a request is only considered valid if $\$\text{fee} \geq F_{\min}$. $F_{\min}$ is defined so that it is high enough to cover gas costs for all of Deliver except the execution of the provided callback. However, callback is only given $\$\text{fee} - F_{\min}$ ether worth of gas to execute. Therefore it is impossible for the entire call of Deliver to spend more than $F_{\min} + (\$\text{fee} - F_{\min}) = \$\text{fee}$ ether on gas.

To prove claim 3 we note that $\$\text{fee} \leq F_{\max}$ and, by construction, $\mathcal{P}_{SGX}$ will always provide at least $F_{\max}$ in gas for the execution of Deliver. Therefore we have that $\mathcal{P}_{SGX}$ will always provide at least \$fee in gas to execute Deliver. By the argument above, Deliver can never use more than \$fee in gas, so therefore an SGX-initialized call to Deliver will never run out of gas. □

Lemma 2. *Any data given as an argument to* callback *in $\mathcal{C}_{TC}$'s Deliver method is verifiably authentic.*

7.2 Other security concerns

We treat side-channel attacks as outside the scope of our initial TC architecture. Such attacks would be of particular concern should the Relay be compromised. Intel explicitly disclaims protections against side-channel attacks in SGX. The ability for the OS to monitor page faults incurred by a process running in an enclave is an

example shown to be potentially serious in practice []. Additionally, the Relay or any network adversary can potentially perform traffic analysis to determine what content the Enclave is retrieving from a remote server [], a potential threat to the confidentiality of private datagrams.

8 Applications

Discuss flight insurance as an example: We'd like to conceal the flight number and date. We might also want to conceal payment, so TC might ingest encrypted addresses and mix them internally.

Micro-loans too? Linkage to Facebook / Keybase.io

9 Advanced Features

9.1 Custom and private datagrams

9.2 Protection against traffic analysis

9.3 Full-blockchain scraping

As a means of reducing communication costs...

9.4 Use of new opcodes

New opcodes would enable processing of fresh attestation

9.5 Protecting against freeloading

9.6 Principled data extraction

Ultimately, we servers might themselves to act as ADFs. Possible migration path: (1) Town Crier; (2) XML labels on data; (3) Integration of Town Crier features into source directly

9.7 Off-chain communication

Mention use of Lamport signatures, etc.

9.8 Revocation

9.9 Migration Path

Ultimately, we expect sources themselves to act as ADFs. The migration path is : (1) Town Crier; (2) XML labels on data; (3) Integration of Town Crier features into source directly

9.10 IoT support

10 Conclusion

A Version 1

Here we define and discuss proposed extensions to the Town Crier protocol.

A.1 Request Cancellation

In order to provide recourse if the system is compromised and disabled or datagrams are delayed beyond a reasonable time, there can be a way to cancel requests for a refund. The refund must withhold a fixed fee of $F_{\min}$ in order to ensure that malicious aborts cannot bankrupt the ADF, but the rest of the fee can be refunded at any time. If the ADF attempts to deliver a datagram for a canceled request, it will simply receive the $F_{\min}$ needed to cover its gas costs for the attempted delivery and not deliver any data.

In order to safely handle request cancellations, we now have to store verification data on the blockchain itself. This will be considerably more expensive, but the Ethereum protocol supports it cleanly. For notational simplicity, we use four blockchain storage functions. They functions create a map from integers to arbitrary data values in the domain V.

- $\mathsf{store} : \mathbb{N} \times V \to \emptyset$. This stores a key with an associated value in the map and returns nothing.
- $\mathsf{load} : \mathbb{N} \to V \cup \{\perp\}$. This returns the value associated with the given key or $\perp$ if the key is not in the map.
- $\mathsf{storeContains} : \mathbb{N} \to \{0,1\}$. This returns whether or not the key exists in the map.
- $\mathsf{remove} : \mathbb{N} \to \emptyset$. This removes the key and its associated value if it is in the map and otherwise does nothing.

Table 1 describes the new $\mathcal{C}_{TC}$ blockchain. The rest of the protocol need to change (save for calls to Deliver requiring slightly different arguments).

Using the same adversarial model, we can make the same guarantees of this new system as we did for the original Town Crier system. Even if a malicious user cancels their request just as Deliver is being called, the cancellation fee is enough to reimburse $\mathcal{P}_{SGX}$ for any gas costs.

If we expand the adversarial model to allow for arbitrary denial of service attacks against the Town Crier system (but not the blockchain), the new Cancel functionality allows affected users to recover most of their fee with no action from the Town Crier system.

$\mathcal{C}_{TC}$ with Cancellation	
Init:	Set $\mathsf{reqs} := \emptyset$ and $\mathsf{reqCnt} := 0$
Request:	Upon receiving $(\mathsf{type}, \mathsf{callback}, \$\mathsf{fee})$ from a user $\mathcal{P}$: If ($\$\mathsf{fee} < F_{\min}$ or $\$\mathsf{fee} > F_{\max}$) Return with no effect. Set $\mathsf{reqID} := \mathsf{reqCnt}$. Set $\mathsf{reqCnt} := \mathsf{reqCnt} + 1$. $\mathsf{store}(\mathsf{reqID} \mapsto (\mathcal{P}, \$\mathsf{fee}))$. Return reqID.

Deliver:	Upon receiving $(\mathsf{reqID}, \mathsf{data}, \mathsf{callback})$ from a user $\mathcal{P}$:
	If $\mathcal{P} \neq \mathcal{P}_{SGX}$
	  Return with no effect.
	If !storeContains(reqID)
	  Send $F_{\min}$ to $\mathcal{P}_{SGX}$.
	  Return with no further effect.
	$(*, \$\mathsf{fee}) \leftarrow \mathsf{load}(\mathsf{reqID})$.
	Call $\mathsf{callback}(\mathsf{data})$ providing $\$\mathsf{fee} - F_{\min}$ ether as the maximum gas.
	Send $fee ether to $\mathcal{P}_{SGX}$.
	$\mathsf{remove}(\mathsf{reqID})$.

Cancel:	Upon receiving (reqID) from a user $\mathcal{P}$:
	If !storeContains(reqID)
	  Return with no effect.
	$(\mathcal{R}, \$\mathsf{fee}) \leftarrow \mathsf{load}(\mathsf{reqID})$.
	If $\mathcal{P} \neq \mathcal{R}$
	  Return with no effect.
	Send $\$\mathsf{fee} - F_{\min}$ to $\mathcal{P}$.
	$\mathsf{remove}(\mathsf{reqID})$.

Table 1: Definition of the $\mathcal{C}_{TC}$ contract with cancellation.

A.2 Service-level Agreements

We start by noting that a service-level agreement (SLA) is generally implemented by paying recompense if it is violated. This seems extreme if the service is not run for a profit, so thus we will assume that the costs of any SLA payments are funded by profits gained when the SLA is not violated. For Town Crier, these profits can be implemented by simply increasing $F_{\min}$ above the gas cost necessary to run Deliver. In this case, the extra money will be profit. Note that if this happens, the cancellation fee could remain simply enough to recoup gas costs and not include the profit.

In this system, and SLA could consist of a maximum amount of time before a datagram is delivered. If the user wishes to cancel a request before that time, it would be considered a voluntary cancellation and incur a cancellation fee high enough to cover gas costs of an attempted delivery. If, however, the SLA has expired before the request is canceled, then not only would a cancellation not incur a fee, the user would be returned their entire initial

fee and an SLA-violation recompense. This could be a small value that would be need to later be paid back by Town Crier system out of the profits from successfully-deliver requests in order to prevent the contract from going bankrupt.

This mechanism presents some danger if a large number of SLAs are violated at the same time and the Town Crier system is unable to provide enough funds to the contract to make all of the recompense payments. In this case, there could be a lightweight function on the contract to inform a user whether or not there is sufficient funding to make an SLA payment. This function could either before cancellation requests or it could be before a request is made. The former case would attempt to guarantee that a cancellation request right now would include an SLA payment. The latter would attempt to ensure that a new request would always have money set aside to pay for an SLA violation. Both of these utility functions may be subject to a race condition of another user making a cancellation or new request between the utility call on the actual call, thus costing an honest user money.

B Basic Protocol

B.1 A Gas-Free Basic Protocol

For simplicity, we first describe a gas-free version of our basic protocol. This basic protocol improves the strawman solution by resolving the aforementioned two issues.

Enclave-specific keys. To avoid having to verify a group signature on the blockchain, during enclave initialization, we have each enclave generate its enclave-specific key pair denoted $(\mathsf{pk}_{\mathrm{TC}}, \mathsf{sk}_{\mathrm{TC}})$. The $\mathsf{sk}_{\mathrm{TC}}$ is retained within the enclave and used to sign the datagrams extracted from data sources during the request phase. Since Ethereum itself already verifies signatures on messages sent from users (i.e., users interact with the Ethereum blockchain through an authenticated channel), we devise a trick to *piggyback the signature verification on top of Ethereum's already existing signature verfication mechanism*. This means that the SGX enclave must sign datagrams using the [elaine: fill in name] signature scheme that is compatible with Ethereum's signature verification. This way, we need not implement a separate signature verification in the user-defined $\mathcal{C}_{TC}$ contract. This saves not only software engineering effort, but more importantly, gas.

To make this idea fully work, in an offline phase, a user must verify an SGX attestation vouching for its own enclave-specific public key $\mathsf{pk}_{\mathrm{TC}}$. This $\mathsf{pk}_{\mathrm{TC}}$ is hardcoded inside the blockchain contract $\mathcal{C}_{TC}$. The user is responsible for making sure that this hardcoded $\mathsf{pk}_{\mathrm{TC}}$ has

User: offline attestation of SGX enclave

Inputs: $\mathsf{pk}_{\mathrm{sgx}}$, $\mathsf{pk}_{\mathrm{TC}}$, $\mathsf{prog}_{\mathrm{encl}}$, σ_{att}

Checks:
Assert $\mathsf{prog}_{\mathrm{encl}}$ is the expected enclave code
Assert $\Sigma_{\mathrm{sgx}}.\mathsf{Verify}(\mathsf{pk}_{\mathrm{sgx}}, \sigma_{\mathrm{att}}, (\mathsf{prog}_{\mathrm{encl}}, \mathsf{pk}_{\mathrm{TC}}))$
Assert $\mathcal{C}_{TC}$ is correct and parametrized w/ $\mathsf{pk}_{\mathrm{TC}}$
// now okay to rely on $\mathcal{C}_{TC}$

Figure 7: A user checks the Town Crier blockchain contract $\mathcal{C}_{TC}$, and verifies an SGX attestation of the enclave's code and its public key $\mathsf{pk}_{\mathrm{TC}}$ before entering a contract that calls $\mathcal{C}_{TC}$. [elaine: here we use a simplified abstraction, but the actual implementation also involves verifying the revocation list.]

an appropriate SGX attestation before interacting with the $\mathcal{C}_{TC}$ blockchain contract.

Instantiating trusted absolute clock. Since SGX's trusted clock provides only relative time with respect to a reference point, we will rely on the following mechanism to realize a trusted *absolute* clock.

- **Offline calibration.** In an offline phase, a user U performs the following calibration protocol with the SGX enclave:

  [elaine: this formal notation needs to be changed, it is not compatible with other formal notation.]

  $\mathcal{U}$: get absolute T_0 from a trusted source
  $\mathcal{U}$: pick random nonce
  $\mathcal{U} \to \mathcal{F}_{\mathrm{sgx}}$: nonce
  $\mathcal{F}_{\mathrm{sgx}} \to \mathcal{U}$: (clock_ref, ΔT_0, nonce)
  $\mathcal{U}$: record clock_ref, ΔT_0

  [elaine: the above assumes that an authenticated channel has been established.]

- **Online trusted absolute clock.** Whenever $\mathcal{F}_{\mathrm{sgx}}$ gives the relative time ΔT with respect to clock_ref, the user *i)* checks that clock_ref agrees with the saved reference point, and *ii)* computes $T_0 + \Delta T - \Delta T_0$ as the absolute time.

Formal protocol description.

B.2 Formal Guarantees

Authenticity. Roughly speaking, authenticity means that an adversary (including a corrupt user, a corrupt relay, or the collusion thereof) cannot convince the Town Crier blockchain contract $\mathcal{C}_{TC}$ to accept a wrong data feed. Here a wrong data feed means any content that

Town Crier blockchain contract $\mathcal{C}_{TC}$

Request: On recv $(\mathsf{id}, \mathsf{params}, \mathsf{callback})$ from some user $\mathsf{pk}_{\mathcal{U}}$:
Record $(\mathsf{id}, \mathsf{params}, \mathsf{callback})$

Deliver: On recv $(\mathsf{id}, \mathsf{params}, \mathsf{data})$ from $\mathsf{pk}_{\mathrm{TC}}$:
Let $(\mathsf{id}, \mathsf{params}', \mathsf{callback})$ be the most recently recorded tuple for id
Assert $\mathsf{params} = \mathsf{params}'$
Call $\mathsf{callback}(\mathsf{data})$

Figure 8: A simple, fee-free version of the Town Crier contract $\mathcal{C}_{TC}$. Note that communication with $\mathcal{C}_{TC}$ is through an authenticated channel implemented through digital signatures (which are not explicitly expressed in our notation).

Town Crier's enclave program

Initialize: On receive "initialize":
$(\mathsf{pk}_{\mathrm{TC}}, \mathsf{sk}_{\mathrm{TC}}) := \Sigma.\mathsf{KeyGen}(1^{\lambda})$
Record $(\mathsf{pk}_{\mathrm{TC}}, \mathsf{sk}_{\mathrm{TC}})$
Output $\mathsf{pk}_{\mathrm{TC}}$ *// incl. in measurement & attestation*

Resume: On receive $(\mathsf{id}, \mathsf{params})$
Parse $\mathsf{params} := (\mathsf{url}, \mathsf{pk}_{\mathrm{url}}, T)$:
$T_{\mathrm{start}} := \mathsf{clock}()$
Establish secure channel w/ url w/ public key $\mathsf{pk}_{\mathrm{url}}$
Download the webpage at url
$T_{\mathrm{end}} := \mathsf{clock}()$
Assert $\mathsf{round}(T_{\mathrm{start}}) = \mathsf{round}(T_{\mathrm{end}}) = T$
Parse webpage and extract data
$\sigma := \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{TC}}, (\mathsf{id}, \mathsf{params}, \mathsf{data}))$
Output $((\mathsf{id}, \mathsf{params}, \mathsf{data}), \sigma)$

Figure 9: SGX enclave's code.

Town Crier's untrusted relay $\mathcal{R}$

Initialize:
Send "initialize" to $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$
On receive $(\mathsf{pk}_{\mathrm{TC}}, \sigma_{\mathrm{att}})$ from $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$:
Publish $(\mathsf{pk}_{\mathrm{TC}}, \sigma_{\mathrm{att}})$

Loop forever:
When $\mathcal{C}_{TC}$ receives new request $(\mathsf{id}, \_, \mathsf{params})$:
Parse $\mathsf{params} := (\mathsf{url}, \mathsf{pk}_{\mathrm{url}}, T)$
Fork:
Wait till time T
Send ("resume", params) to $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$
On recv $((\mathsf{id}, \mathsf{params}, \mathsf{data}), \sigma)$ from $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$:
AuthSend $(\mathsf{id}, \mathsf{params}, \mathsf{data})$ to $\mathcal{C}_{TC}$

Figure 10: Town Crier untrusted relay. For simplicity, here we assume that there is only a single enclave program. When multiple data feed sources are supported, we need multiple enclaves that instantiate different parsers for different sites. In this case, the Town Crier relay also initialize all enclave instances and route the request to the correct enclave instance.

$\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$: abstraction for SGX

Hardcoded: $\mathsf{sk}_{\mathrm{sgx}}$

Assume:
$\mathsf{prog}_{\mathrm{encl}}$ has entry points **Initialize** and **Resume**

Initialize:
On receive "initialize" from $\mathcal{R}$:
Let $\mathsf{outp} := \mathsf{prog}_{\mathrm{encl}}.\mathbf{Initalize}()$
$\sigma_{\mathrm{att}} := \Sigma_{\mathrm{sgx}}.\mathsf{Sign}(\mathsf{sk}_{\mathrm{sgx}}, (\mathsf{prog}_{\mathrm{encl}}, \mathsf{outp}))$
// models group sig.
Output $(\mathsf{outp}, \sigma_{\mathrm{att}})$

On receive ("resume", params) from $\mathcal{R}$:
Let $\mathsf{outp} := \mathsf{prog}_{\mathrm{encl}}.\mathbf{Resume}(\mathsf{params})$
Output outp

Figure 11: Formal abstraction for SGX attested execution. We adopt a similar modeling approach by Shi et al., where the SGX group signature is abstracted with a normal signature by a manufacturer key $\mathsf{pk}_{\mathrm{sgx}}$. [elaine: cite our sok paper] The above functionality only models a subset of SGX features that is sufficient for our formalism.

Program for Town Crier Relay $\mathcal{R}$

Initialize: Same as Figure 13
Loop forever:
Whenever $\mathcal{C}_{TC}$ receives $(\mathsf{id}, \mathsf{params}, \_, \_, \Delta\mathrm{F}_{\mathrm{request}} + \$\mathrm{F}_{\mathrm{deliver}})$:
  Send $(\texttt{resume}, (\mathsf{id}, \mathsf{params}))$ to $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$
  On recv $((\mathsf{id}, \mathsf{params}, \mathsf{data}), \sigma)$ from $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$:
    Send $((\mathsf{id}, \mathsf{params}, \mathsf{data}), \sigma)$ to $\mathcal{C}_{TC}$ // **msg.** m_3

Figure 13: The Town Crier Relay $\mathcal{R}$.

Town Crier blockchain contract $\mathcal{C}_{TC}$ with fees

Request: On recv $(\mathsf{id}, \mathsf{params}, \mathsf{callback}, \mathrm{T}_{\mathrm{timeout}}, \Delta\mathrm{F}_{\mathrm{request}} + \$\mathrm{F}_{\mathrm{deliver}})$ from some user $\mathsf{pk}_{\mathcal{U}}$:
Assert $\mathrm{T}_{\mathrm{timeout}} > \mathsf{cur\_time}$
Record $(\mathsf{id}, \mathsf{params}, \mathsf{callback}, \$\mathrm{F}_{\mathrm{deliver}}, \mathrm{T}_{\mathrm{timeout}}, \mathsf{pk}_{\mathcal{U}})$
// *at most* $\Delta F_{request}$ *gas consumed*
// *all remaining gas returned to* $\mathsf{pk}_{\mathcal{U}}$
// $\$F_{deliver}$ *held by contract*

Deliver: On recv $(\mathsf{id}, \mathsf{params}, \mathsf{data}, \Delta\mathrm{F}_{\mathrm{deliver}})$ from $\mathsf{pk}_{\mathrm{TC}}$:
Let $(\mathsf{id}, \mathsf{params}', \mathsf{callback}, \$\mathrm{F}'_{\mathrm{deliver}}, \mathrm{T}_{\mathrm{timeout}}, \_)$ be the most recently recorded tuple for id
Assert $\mathrm{T}_{\mathrm{timeout}} > \mathsf{cur\_time}$
Assert $\mathsf{params} = \mathsf{params}'$
Assert $\$\mathrm{F}'_{\mathrm{deliver}} \leq \Delta\mathrm{F}_{\mathrm{deliver}}$
Send $\$\mathrm{F}'_{\mathrm{deliver}}$ to $\mathsf{pk}_{\mathrm{TC}}$
Call $\mathsf{callback}(\mathsf{data})$
// *at most* $\Delta F_{deliver}$ *gas consumed*
// *all remaining gas returned to* pk_{TC}

Cancel: On recv $(\mathsf{id}, \Delta\mathrm{F}_{\mathrm{cancel}})$ from some user $\mathsf{pk}_{\mathcal{U}}$
Let $(\mathsf{id}, \_, \_, \$\mathrm{F}_{\mathrm{deliver}}, \mathrm{T}_{\mathrm{timeout}}, \mathsf{pk}'_{\mathcal{U}})$ be the most recently recorded tuple for id
Assert $\mathsf{pk}_{\mathcal{U}} = \mathsf{pk}'_{\mathcal{U}}$
Assert $\mathsf{cur\_time} \geq \mathrm{T}_{\mathrm{timeout}}$
Send $\$\mathrm{F}_{\mathrm{deliver}}$ to $\mathsf{pk}_{\mathcal{U}}$

Figure 12: Town Crier contract $\mathcal{C}_{TC}$ reflecting fees. $\Delta\mathrm{F}_{\mathrm{request}}$ denotes the gas for executing the **Request** entry point. $\Delta\mathrm{F}_{\mathrm{deliver}}$ denotes the gas for executing the **Deliver** entry point that includes the user-defined callback. $\$\mathrm{F}_{\mathrm{deliver}}$ denotes $\mathrm{F}_{\mathrm{deliver}}$ amount of explicit, non-gas currency units. Essentially, the requester first pays $\$\mathrm{F}_{\mathrm{deliver}}$ currency units which will be used to refund the $\Delta\mathrm{F}_{\mathrm{deliver}}$ amount of gas that $\mathsf{pk}_{\mathrm{TC}}$ will need to put in to call the **Deliver** entry point.

differs from the expected content obtained by crawling the specified url at the specified time T.

In formally defining authenticity, we assume that the user and the blockchain contract $\mathcal{C}_{TC}$ behave honestly. Recall that the user must verify upfront the attestation σ_{att} that vouches for the enclave's public key $\mathsf{pk}_{\mathrm{TC}}$.

[elaine: the proof may need to be fixed since the protocol was modified.]

Definition 1 (Authenticity)**.** *We say that the Town Crier protocol satisifies authenticity of data feed, if for any polynomial-time adversary that can interact arbitrarily with $\mathcal{F}_{sgx}$, it cannot persuade an honest verifier to accept a tuple* $(\mathsf{pk}_{TC}, \sigma_{att}, \mathsf{params} := (\mathsf{url}, \mathsf{pk}_{url}, T), \mathsf{data}, \sigma)$ *where* data *is not the contents of* url *with the public key* pk_{url} *at time T. More formally, for any probablistic polynomial-time adversary $\mathcal{A}$*

$$\Pr\left[\begin{array}{l}(\mathsf{pk}_{TC}, \sigma_{att}, \mathsf{id}, \mathsf{params}, \mathsf{data}, \sigma) \leftarrow \mathcal{A}^{\mathcal{F}_{sgx}}(1^{\lambda}) : \\ \quad \left(\Sigma_{sgx}.\mathsf{Verify}(\mathsf{pk}_{\mathrm{sgx}}, \sigma_{att}, (\mathsf{prog}_{encl}, \mathsf{pk}_{TC})) = 1\right) \wedge \\ \quad (\Sigma.\mathsf{Verify}(\mathsf{pk}_{TC}, \mathsf{id}, \mathsf{params}, \mathsf{data}) = 1) \wedge \\ \quad \mathsf{data} \neq \mathsf{prog}_{encl}(\mathsf{params})\end{array}\right] \leq \mathsf{negl}(\lambda)$$

Theorem 1 (Authenticity)**.** *Assume that Σ_{sgx} and Σ are secure signature schemes (recall that we follow Shi et al. [elaine: cite] who show how to abstractly model SGX's group signature as a regular signature scheme under a manufacturer public key* $\mathsf{pk}_{\mathrm{sgx}}$*), then, the above protocol achieves authenticity of data feed by Definition 1.*

Proof. (sketch.) We show that if the adversary $\mathcal{A}$ succeeds in a forgery with non-negligible probability, we can construct an adversary $\mathcal{B}$ that can either break Σ_{sgx} or Σ with non-neligible probability. We consider two cases. The reduction $\mathcal{B}$ will flip a random coin to guess which case it is, and if the guess is wrong, simply abort.

- Case 1: $\mathcal{A}$ outputs a signature σ that uses the same $\mathsf{pk}_{\mathrm{TC}}$ as the SGX functionality $\mathcal{F}_{\mathrm{sgx}}$. In this case, $\mathcal{B}$ will try to break Σ. $\mathcal{B}$ interacts with a signature challenger Ch who generates some $(\mathsf{pk}^*, \mathsf{sk}^*)$ pair, and gives to $\mathcal{B}$ the public key pk^*. $\mathcal{B}$ simulates $\mathcal{F}_{\mathrm{sgx}}$ by

implicitly letting $\mathsf{pk}_{\text{TC}} := \mathsf{pk}^*$. Whenever $\mathcal{F}_{\text{sgx}}$ needs to sign a query, $\mathcal{B}$ passes the signing query onto the signature challenger Ch.

Since $\mathsf{data} \neq \mathsf{prog}_{\text{encl}}(\mathsf{params})$, $\mathcal{B}$ cannot have queried Ch on a tuple of the form $(\_, \mathsf{params}, \mathsf{data})$. Therefore, $\mathcal{B}$ simply outputs what $\mathcal{A}$ outputs (suppressing unnecessary terms) as the signature forgery.

- Case 2: $\mathcal{A}$ outputs a signature σ that uses a different pk_{TC} as the SGX functionality $\mathcal{F}_{\text{sgx}}$. In this case, $\mathcal{B}$ will seek to break Σ_{sgx}. $\mathcal{B}$ interacts with a signature challenger Ch, who generates some $(\mathsf{pk}^*, \mathsf{sk}^*)$ pair, and gives to $\mathcal{B}$ the public key pk^*. $\mathcal{B}$ simulates $\mathcal{F}_{\text{sgx}}$ by implicitly setting $\mathsf{pk}_{\text{sgx}} := \mathsf{pk}^*$. Whenever $\mathcal{F}_{\text{sgx}}$ needs to make a signature with sk_{sgx}, $\mathcal{B}$ simply passes the signature query onto Ch. In this case, in order for $\mathcal{A}$ to succeed, it must produce a valid signature σ_{att} for a different public key pk'. Therefore, $\mathcal{B}$ simply outputs this as a signature forgery.

□

C Extensions

C.1 Handling Transaction Fees

To mitigate potential Denial-of-Service (DoS) attacks, Ethereum employs a fee mechanism, referred to as "gas", where the submitter of a transaction (that invokes an entry point in the contract) pays a transaction fee roughly proportional to the execution time of the corresponding entry point.

Notations and assumed execution model. In Figure [elaine: fill], we use the notation $\Delta\mathtt{F}$ to denote transaction fees (i.e., gas), where Δ is a type annotation and $\mathtt{F}$ denotes the numerical amount of the gas. Other non-gas, normal currency units are denoted as $\$\mathtt{F}$ where $\$$ is a type annotation, and $\mathtt{F}$ denotes the amount of the currency. For simplicity, our notational system assumes that gas and normal currency adopt the same currency unit.

We assume that the blockchain contract adopts the following execution model for gas which closely resembles Ethereum's execution model:

- *Providing gas.* When a transaction is submitted, it invokes an entry point in the contract. The transaction submitter provides a gas amount to activate the entry point.
- *Extra gas.* If extra gas remains at the end of the execution (after invoking an entry point), all extra gas is refunded to the transaction submitter at the end.
- *Gas exhaustion.* Gas exhaustion is dealt with in the following manner. Consider each entry point of the contract as a function. Functions can call other functions. Each function can specify a gas upper bound not to exceed the remaining gas of the parent function (and if left unspecified, the upper bound is implicitly set to all remaining gas of the parent function). If execution of the function exhausted the per-function gas upper bound, the function execution is aborted and state reverted to before the function is invoked.

Town Crier protocol with transaction fees. Our basic Town Crier implements a policy where the requester pays for all gas needed and Town Crier in effect pays nothing. We now describe how this can be realized by modifying the fee-free protocol described in Section [elaine: refer]. [elaine: need to add time to all the description below.]

- *Initialization.* To initialize the system, we assume that Town Crier deposits a fixed amount $\Delta\mathtt{F}_{\text{max}}$ into the wallet account pk_{TC}.
- *Town Crier blockchain contract.* Figure [elaine: refer] describes the Town Crier blockchain contract reflecting fees. Since Town Crier's account pk_{TC} has to invoke the **Deliver** entry point, it has to advance a gas payment $\Delta\mathtt{F}_{\text{deliver}}$. This amount will be entirely refunded through money deposited in the contract by the requester.
- *Town Crier Relay.* The Town Crier relay monitors the blockchain, and whenever the blockchain contract $\mathcal{C}_{TC}$ receives a new request $(\mathsf{id}, \mathsf{params}, \mathsf{callback}, \Delta\mathtt{F}_{\text{request}} + \$\mathtt{F}_{\text{deliver}})$, it asserts that
$$\Delta\mathtt{F}_{\text{min}} \leq \$\mathtt{F}_{\text{deliver}} \leq \Delta\mathtt{F}_{\text{max}}$$
where $\Delta\mathtt{F}_{\text{max}}$ is the total amount of money in Town Crier's account pk_{TC}, and $\Delta\mathtt{F}_{\text{min}}$ is the cost of executing the **Deliver** entry point when the user-defined callback is empty. The check $\$\mathtt{F}_{\text{deliver}} \leq \Delta\mathtt{F}_{\text{max}}$ ensures that Town Crier's enclave has sufficient funds to advance for the **Deliver** phase. The check $\Delta\mathtt{F}_{\text{min}} \leq \$\mathtt{F}_{\text{deliver}}$ ensures that the **Deliver** entry point should have sufficient gas to execute everything excluding the user-defined callback – this guarantees that the statement where Town Crier gets refunded for the gas is always reached.

  Finally, the Town Crier relay passes the tuple $(\mathtt{resume}, (\mathsf{id}, \mathsf{params}, \Delta\mathtt{F}_{\text{deliver}} = \$\mathtt{F}_{\text{deliver}}))$ as input to the enclave.
- *Town Crier enclave.* We make the following small modification to the fee-free protocol described in Figure [elaine: refer]. Instead of signing

the tuple $(\mathsf{id}, \mathsf{params}, \mathsf{data})$ at the end of the enclave's execution, the enclave now signs the tuple $(\mathsf{id}, \mathsf{params}, \mathsf{data}, \Delta\mathrm{F}_{\mathrm{deliver}})$ instead, where signing $\Delta\mathrm{F}_{\mathrm{deliver}}$ authorizes a gas amount of $\Delta\mathrm{F}_{\mathrm{deliver}}$ to be advanced to the contract (which will be refunded later).

- *Requester.* The honest requester would behave the same way as in Figure [elaine: refer], except for additionally putting in $\Delta\mathrm{F}_{\mathrm{request}} + \$\mathrm{F}_{\mathrm{deliver}}$ amount with each request, where the honest requester would set $\Delta\mathrm{F}_{\mathrm{request}}$ to the gas cost of executing the **Request** entry point, and set $\$\mathrm{F}_{\mathrm{deliver}}$ to be the cost of executing the **Deliver** entry point (including the cost of executing the user-defined callback function).

Theorem 2 (Gas neutrality for Town Crier)**.** *Assuming that the Town Crier relay is honest, then Town Crier's wallet account* pk_{TC} *will have at least* $\Delta\mathrm{F}_{max}$ *amount remaining after each* **Deliver** *call finishes execution.*

(sketch). Since the relay is honest, every time $\mathsf{pk}_{\mathrm{TC}}$ invokes the **Deliver** entry point, the following holds: 1) $\$\mathrm{F}_{\mathrm{deliver}} = \Delta\mathrm{F}_{\mathrm{deliver}}$; i.e., the gas $\mathsf{pk}_{\mathrm{TC}}$ advances is equal to the fees the requester deposited with the $\mathcal{C}_{TC}$ contract; and 2) the amount gas sent $\Delta\mathrm{F}_{\mathrm{deliver}} = \$\mathrm{F}_{\mathrm{deliver}} \geq \Delta\mathrm{F}_{\mathrm{min}}$; in other words, the statement that refunds $\mathsf{pk}_{\mathrm{TC}}$ $\$\mathrm{F}_{\mathrm{deliver}}$ amount will definitely be invoked. This ensures that the full gas amount $\Delta\mathrm{F}_{\mathrm{deliver}}$ that $\mathsf{pk}_{\mathrm{TC}}$ advanced will be refunded (and anything more left at the end of the execution will also be refunded). □

Theorem 3 (Bounded loss for honest user)**.** *[elaine: fill in]*