Town Crier:
An Authenticated Data Feed for Smart Contracts

Fan Zhang
Cornell Univ.

Ethan Cecchetti
Cornell Univ.

Kyle Croman
Cornell Univ.

Ari Juels
Cornell Tech (Jacobs)

Elaine Shi
Cornell Univ.

18 February 2016
v1.0

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 a stock price.) Trustworthy *data feeds* that can support data requests by 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. It uses trusted hardware to authenticate and scrape data from HTTPS-enabled websites and to generate trustworthy data for relying smart contracts. It also includes a range of advanced features such as support for private data requests, which involve decryption and evaluation of request ciphertext 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 Intel SGX software development kit and furnishes data for the smart-contract system Ethereum. To the best of our knowledge, ours is the first research paper reporting system implementation on a real SGX-enabled host. Finally, we 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

Smart contracts are computer programs that autonomously execute the terms of a contract. For decades they have been envisioned as a way to use logical specification to render legal agreements more precise, pervasive, and efficiently executable. Szabo, who popularized the term "smart contact" in a seminal 1994 essay [33], gave as an example a smart contract that enforces car loan payments. If the owner of the car fails to make a timely payment, a smart contract could programmatically revoke physical access and return control of the car to the bank.

Cryptocurrencies such as Bitcoin [28] have provided key technical provisions for decentralized smart contracts: direct control of money by programs and fair, automated execution of computations through the decentralized consensus mechanisms underlying blockchains. The recently launched Ethereum supports Turing-complete code and thus fully expressive, self-enforcing smart contracts, a big step toward the vision of researchers and proponents.

As Szabo's example shows, however, the most compelling applications of smart contracts require not just blockchain code, but access to data about real-world state and events. Similarly, financial contracts and derivatives, key applications for Ethereum [15,35], rely on data about financial markets such as equity prices.

Data feeds—contracts on the blockchain that serve data requests by other contracts [15, 35]—are intended to meet this need. A few data feeds exist for Ethereum today, but provide no assurance of trustworthy data beyond the reputation of their operators (who are typically individuals or small entities), even if their data originates with trustworthy sources. Of course, there exist reputable websites that serve data for non-blockchain applications and use HTTPS, enabling source authentication of served data. Smart contracts, though, lack network access and thus cannot directly access such data. The lack of a substantive ecosystem of trustworthy data feeds thus remains an oft-cited, critical obstacle to the evolution of in Ethereum and smart contracts in general [1].

Town Crier. We introduce a system called *Town Crier* (TC) that provides an *authenticated data feed* (ADF) for smart contracts. TC acts as a high-trust bridge between existing HTTPS-enabled data websites and the Ethereum blockchain. It retrieves website data and serves it to re-

lying contracts on the blockchain as concise, contract-consumable pieces of data (e.g., stock quotes) called *datagrams*. TC makes use of Software Guard Instructions (SGX), Intel's recently released trusted hardware capability. It executes its core functionality as a trusted piece of code in an SGX *enclave*, which protects against malicious processes and the OS and can *attest* (prove) to a remote client that the client is interacting with a legitimate, SGX-backed instance of the TC code.

Through a smart-contract front end, Town Crier responds to requests by contracts on the blockchain with attestations of the following form:

> "Datagram X specified by parameters params is served by an HTTPS-enabled website Y during a specified time frame T."

A relying contract can be assured of the correctness of X in such a datagram if it trusts the security of SGX, the (published) TC code, and the validity of source data in the specified interval of time.

Contributions of TC. We highlight several important contributions in our design of TC:

Fully functional TC implementation, with pending open source and launch. We designed and implemented Town Crier as a complete, end-to-end system that offers formal security guarantees at the cryptographic protocol level. Aiming beyond an advance in academic research, we plan to launch Town Crier as an open-source, production service atop Ethereum in the near future. Our launch of TC awaits only availability of the Intel Attestation Service (IAS), which is expected to occur soon. In its initial form, Town Crier be a free service for smart contract users, requiring users only to defray the (small) cost of invoking TC on the Ethereum blockchain.

Formal security analysis. Formal security is vitally important to a data feed. Smart contracts execute in an adversarial environment where parties can reap financial gains by subverting the contracts or the services on which they rely. Legal recourse is often impractical precisely because of several benefits smart contracts provide; they enable micro-services without costly legal setup and enforcement and allow contracts between arbitrary pseudonymous parties. We thus adopt a rigorous, principled approach to the design of Town Crier by formally defining and ensuring:

- *Authenticity*: A datagram X returned to a requesting contract is guaranteed to truly reflect the data served by specified website Y in time interval T with the requester's specified parameters params.
- *Gas neutrality (zero TC loss).* Assuming that TC executes honestly, it does not lose money (cannot suffer resource depletion) even when accessed by arbitrarily malicious blockchain contracts and users.
- *Fair expenditure (bounded requester loss).* Even when all other users, contracts, and TC itself act maliciously, an honest requester will never pay more than a tiny amount beyond what is required for valid computation executed for that request—whether or not a datagram is delivered.

To obtain the above formal guarantees, we rely on the formal modeling of blockchains proposed by Kosba et al. [26] and the formal abstraction for SGX proposed by Shi et al. [32] Our analysis of TC reveals interesting challenges and technical subtleties, e.g., a subtle gap between the formal blockchain model of Kosba et al. [26] and Ethereum's instantiation that proves important in formal reasoning about TC's handling of fees.

Robustness to component compromise. TC minimizes the Trusted Computing Base (TCB) of its trusted code in the SGX enclave. It thus offers a basic security model in which a user need only trust SGX itself and a designated data source (website). As an additional feature, TC can hedge against the risk of compromise of a website or single SGX instance by supporting various modes of majority voting: among multiple websites offering the same piece of data (e.g., stock price), or among multiple, possibly geographically dispersed, SGX instances.

Private and custom datagrams. To meet the potentially complex confidentiality concerns that can arise in the broad array of smart contracts enabled by TC, TC's trusted enclave code is instrumented to ingest confidential user data (encrypted under a TC public key). It can thereby support *private* datagram requests, with encrypted parameters, and *custom* datagram requests, which securely access the online resources of requesters (e.g., online accounts) using encrypted credentials.

Additionally, to the best of our knowledge, ours is the first research paper reporting implementation of a substantive system on a real, SGX-enabled host, as opposed to an emulator (e.g., [11, 30]).

Applications. Thanks to the above key contributions, we believe that TC can spur deployment of a rich spectrum of smart contracts that are hard to realize in the existing Ethereum ecosystem. We present three examples that showcase TC's capabilities and demonstrate its end-to-end use: (1) A financial derivative (cash-settled put option) that consumes stock ticker data; (2) A flight insurance contract that relies on private data requests about flight cancellations; and (3) A contract for sale of virtual goods and online games (via Steam Marketplace) for ether, the Ethereum currency, using custom data requests to access online user accounts. We experimentally measure response times for associated datagram requests

ranging from 192-1309 ms, depending on the datagram type. These times are significantly less than an Ethereum block interval, and suggest that a few SGX-enabled hosts can support TC data feed rates well beyond the global transaction rate of a modern decentralized blockchain.

Organization: We present basic technical background for TC (Section 2), followed by an architectural description (Section 3), a basic set of protocols (Section 4), the full, enhanced system protocols (Section 5), and a formal security analysis (Section 6). We then present three example applications (Section 7) which we use as the basis for performance evaluations of TC (Section 8). After presenting related work (Section 9), we conclude the paper (Section 10). The paper appendix includes implementation details, formal modeling and proofs, and future directions omitted from the paper body.

2 Background

In this section, we provide basic background on the main technologies TC incorporates, namely SGX, TLS / HTTPS, and smart contracts.

SGX. Intel's Software Guard Extensions (SGX) [9, 23, 27, 29] is a set of new instructions that confer hardware protections on user-level code. SGX enables process execution in a protected address space known as an *enclave*. The enclave protects the confidentiality and integrity of the process from certain forms of hardware attack and other software on the same host, including the operating system.

A enclave process cannot make system calls, but can read and write memory outside the enclave region. 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]

SGX 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 supplementary data provided by the process, such as a public key. The report is digitally signed 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*, can be verified by a remote system, while the process-provided public key can 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 any client that wishes to verify an attestation. SGX signs quotes using a *group signature* scheme called EPID [13]. This choice of primitive is significant in our design of Town Crier, as EPID is a proprietary signature scheme 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 indexed by a nonce. A reference point remains stable, but SGX does not provide a source of absolute or wall-clock time, another limitation we must work around in TC.

TLS / HTTPS. We assume basic familiarity by readers with TLS and HTTPS (HTTP over TLS). As we explain later, TC exploits an important feature of HTTPS, namely that it can be partitioned into interoperable layers: an HTTP layer interacting with web servers, a TLS layer handling handshakes and secure communication, and a TCP layer providing reliable data stream.

Smart contracts. While TC can in principle support any smart-contract system, we focus in this paper on its use in Ethereum, whose model we now explain. For further details, see [15, 35]

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. A contract accepts messages as inputs to any of a number of designated functions. These entry points, determined by the contract creator, represent the API of the contract. Once created, a contract executes autonomously; it persists indefinitely with even its creator unable to modify its code.[2] 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 a shortfall in gas occurs, as explained below). The "poking" model aside, as a simple abstraction, a smart contract may be viewed as an *autonomous agent* on the blockchain.

Ethereum has its own associated cryptocurrency called *ether*. (At the time of writing, 1 ether has a market value of just over $5 U.S. [2].) To prevent Denial-of-Service (DoS) attacks, prevent inadvertent infinite looping within contracts, and generally control network resource expen-

[1] This model is a simplification: SGX is known to expose some internal enclave state to the OS [20]. Our basic security model for TC assumes ideal isolated execution, but again, TC can also be distributed across multiple SGX instances as a hedge against compromise.

[2] There is one exception: a special opcode `suicide` wipes code from a contract account.

diture, Ethereum allows ether-based purchase of a resource called *gas* to power contracts. Every operation, including sending data, executing computation, and storing data, has a fixed gas cost. Transactions include a parameter (`GASLIMIT`) specifying a bound on the amount of gas expended by the computations they initiate. When a function calls another function, it may optionally specify a lower `GASLIMIT` for the child call which expends gas from the same pool as the parent. Should a function fail to complete due to a gas shortfall, it is aborted and any state changes induced by the partial computation are rolled back to their pre-call state; previous computations on the call path, though, are retained.

Along with a `GASLIMIT`, a transaction specifies a `GASPRICE`, 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 `GASLIMIT` $\times$ `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 of TC's Ethereum front end, an attacker could exhaust the ether used to power the delivery of datagrams.

Finally, we note that transactions in Ethereum are digitally signed for a wallet using ECDSA on the curve Secp256k1 and the hash function SHA3-256.

3 TC Architecture and Security Model

Town Crier includes three main components: The TC Contract ($\mathcal{C}_{TC}$), the Enclave (whose code is denoted by $\mathsf{prog}_{\mathsf{encl}}$), and the Relay ($\mathcal{R}$). The Enclave and Relay reside on the TC server, while the TC Contract resides on the blockchain. We refer to a smart contract making use of the Town Crier service as a *requester* or *relying* contract, which we denote $\mathcal{C}_U$, and its (off-chain) owner as a *client* or *user*. A *data source*, or *source* for short, is an online server (running HTTPS) that provides data which TC draws on to compose datagrams.

An architectural schematic of TC showing its interaction with external entities is given in Figure 1.

The TC Contract $\mathcal{C}_{TC}$. The TC Contract is a smart contract that acts as the blockchain front end for TC. It is designed to present a simple API to a relying contract $\mathcal{C}_U$ for its requests to TC. $\mathcal{C}_{TC}$ accepts datagram requests from $\mathcal{C}_U$ and returns corresponding datagrams from TC. Additionally, $\mathcal{C}_{TC}$ manages TC's monetary resources.

The Enclave. We refer to an instance of the TC code running in an SGX enclave simply as the Enclave and denote the code itself by $\mathsf{prog}_{\mathsf{encl}}$. 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

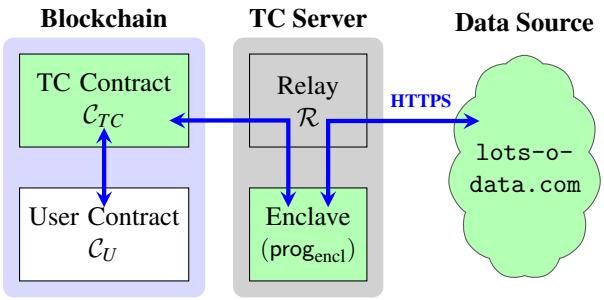

Figure 1: **Basic Town Crier architecture.** Trusted components are depicted in green.

services. It returns a datagram to a requesting contract $\mathcal{C}_U$ as a digitally signed blockchain message. Under our basic security model for SGX, network functions aside, 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 SGX 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 websites) 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 to cause 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:

- *The TC Contract.* $\mathcal{C}_{TC}$ is globally visible on the blockchain and its source code is published for clients. Thus we assume that $\mathcal{C}_{TC}$ behaves honestly.
- *Data sources.* We assume that clients trust the data sources from which they obtain TC datagrams. We also assume that these sources are stable, i.e., yield consistent datagrams, during a requester's specified time interval T. (Requests are generally time-invariant, e.g., for a stock price at a particular time.)
- *Enclave security.* We make three assumptions: (1) The Enclave behaves honestly, i.e., $\mathsf{prog}_{\text{encl}}$, whose source code is published for clients, correctly executes the TC protocol; (2) For an Enclave-generated keypair $(\mathsf{sk}_{TC}, \mathsf{pk}_{TC})$, the private key sk_{TC} is known only to the Enclave; and (3) The Enclave has an accurate (internal) real-time clock. We explain below how we use SGX to achieve these properties.
- *Blockchain communication.* Transaction and message sources are authenticable, i.e., a transaction m sent from an account / wallet $\mathcal{W}_X$ (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.
- *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 Basic 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 $\mathsf{prog}_{\text{encl}}$ and then returns the response 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 operation of 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, deferring this delicate aspect of the system design to Section 5.

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 $m_1 = (\mathsf{params}, \mathsf{callback})$ to $\mathcal{C}_{TC}$ on the blockchain. params specifies the requested datagram, e.g. $\mathsf{params} := (\mathsf{url}, \mathsf{spec}, T)$, where url is the target data source, spec specifies content of a the datagram to be retrieved (e.g., a stock ticker at a particular time), and T specifies the delivery time for the datagram (initiated by scraping of the data source). The parameter $\mathsf{callback}$ in m_1 indicates the entry point in $\mathcal{C}_U$ to which the datagram is to be returned. (In principle, $\mathsf{callback}$ could point to a different contract, but TC does not yet adopt this generalization.)

$\mathcal{C}_{TC}$ generates a fresh unique id and 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 (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 $\mathsf{callback}$ in $\mathcal{C}_U$ in message m_4.

For simplicity here, we assume that $\mathcal{C}_U$ makes a one-time datagram request. Thus it can trivially match m_4 with m_1. Our full protocol contains an optimization by which $\mathcal{C}_{TC}$ returns id to $\mathcal{C}_U$ after m_1 as a consistent, trustworthy identifier for all data flows. This enables straightforward handling of multiple datagram requests from the same instance of $\mathcal{C}_U$.

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.

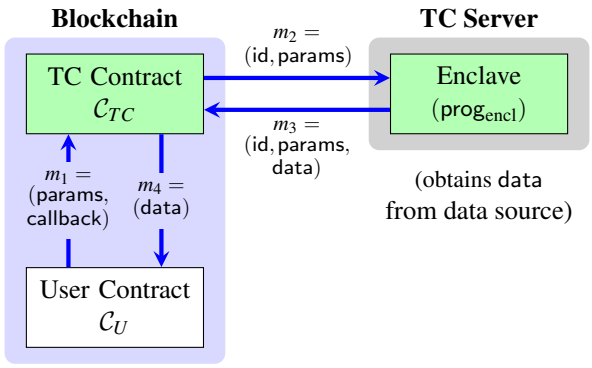

Figure 2: **Data flows in datagram processing.**

Digital signatures are needed to authenticate messages, such as m_3, entering the blockchain from an external source. We let $(\mathsf{sk}_{TC}, \mathsf{pk}_{TC})$ denote the private / public keypair associated with the Enclave for such message authentication. For simplicity, Fig. 2 assumes that the Enclave can send signed messages directly to $\mathcal{C}_{TC}$. We explain shortly how Ethereum requires a layer of indirection such that TC sends m_3 as a transaction via an Ethereum wallet $\mathcal{W}_{TC}$.

4.3 Use of SGX

Let $\mathsf{prog}_{\mathrm{encl}}$ represent the code for Enclave, which we presume is trusted by all system participants. Our protocols in TC rely on the ability of SGX to attest to execution of an instance of $\mathsf{prog}_{\mathrm{encl}}$ and bind a public key pk_{TC} to this instance. Here we briefly explain how we achieve these goals. First, we present a model that abstracts away implementation details in SGX, helping simplify our protocol presentation and later our security proofs. We then explain how SGX attestation is used to authenticate datagrams served by $\mathcal{C}_{TC}$, namely through binding of pk_{TC} to an Ethereum wallet on the blockchain. Finally, we explain how we use the clock in SGX. Our discussion draws on formalism for SGX from [32].

Formal model and notation. We adopt a formal abstraction of Intel SGX proposed by Shi et al. [32]. Following the UC and GUC paradigms [16–18], Shi et al. propose to abstract away the details of SGX implementation, and instead view SGX as a third party trusted for both confidentiality and integrity. Specifically, we use a global UC functionality $\mathcal{F}_{\mathrm{sgx}}(\Sigma_{\mathrm{sgx}})[\mathsf{prog}_{\mathrm{encl}}, \mathcal{R}]$ to denote (an instance of) an SGX functionality parameterized by a (group) signature scheme Σ_{sgx}. Here $\mathsf{prog}_{\mathrm{encl}}$ denotes the SGX enclave program and $\mathcal{R}$ the physical SGX host (which we assume for simplicity is the same as that for the TC Relay). As described in Fig. 3, upon initialization, $\mathcal{F}_{\mathrm{sgx}}$ runs $\mathsf{outp} := \mathsf{prog}_{\mathrm{encl}}.\textbf{Initialize}()$ and attests to the code of $\mathsf{prog}_{\mathrm{encl}}$ as well as outp. Upon a resume call with $(\mathsf{id}, \mathsf{params})$, $\mathcal{F}_{\mathrm{sgx}}$ runs and outputs the result of $\mathsf{prog}_{\mathrm{encl}}.\textbf{Resume}(\mathsf{id}, \mathsf{params})$. Further formalism for $\mathcal{F}_{\mathrm{sgx}}$ is given in Appendix B.1.

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

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

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

Initialize:

On receive (`init`) from $\mathcal{R}$:
- Let $\mathsf{outp} := \mathsf{prog}_{\mathrm{encl}}.\textbf{Initalize}()$
- *// models EPID signature.*
- $\sigma_{\mathrm{att}} := \Sigma_{\mathrm{sgx}}.\mathsf{Sign}(\mathsf{sk}_{\mathrm{sgx}}, (\mathsf{prog}_{\mathrm{encl}}, \mathsf{outp}))$
- Output $(\mathsf{outp}, \sigma_{\mathrm{att}})$

Resume:

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

Figure 3: Formal abstraction for SGX execution capturing a subset of SGX features sufficient for implementation of TC.

Binding $\mathsf{prog}_{\mathbf{encl}}$ to Ethereum wallet $\mathcal{W}_{TC}$. 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 from the Enclave to $\mathcal{C}_{TC}$ is through a wallet $\mathcal{W}_{TC}$. 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 $\mathcal{W}_{TC}$ 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. 4 gives the protocol for a client to check that $\mathcal{C}_{TC}$ is backed by a valid Enclave instance. (We omit modeling here of IAS online revocation checks.)

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

SGX Clock. As noted above, the trusted clock for SGX provides only relative time with respect to a reference point. To work around this, the Enclave is initialized

User: offline verification of SGX attestation

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

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

Figure 4: A client checks an SGX attestation on the enclave's code $\mathsf{prog}_{\text{encl}}$ and public key pk_{TC}. The client also checks that pk_{TC} is hardcoded into TC blockchain contract $\mathcal{C}_{TC}$ before using $\mathcal{C}_{TC}$.

with the current wall-clock time provided by a trusted source, e.g. the Relay (under a trust-on-first-use model). In the current implementation of TC, clients may, in real time, request and verify a fresh timestamp—signed by the Enclave under pk_{TC}—via a web interface in the Relay. Thus, a client can determine the absolute clock time of the Enclave to a degree of accuracy bounded by the round-trip time of its attestation request plus the attestation verification time—hundreds of milliseconds in a wide-area network. This high degree of accuracy is potentially useful for some applications but only loose accuracy is required for most. Ethereum targets a block interval of 12 s 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. For simplicity, we assume in our protocol specifications that the Enclave clock provides accurate wall-clock time in the canonical format of seconds since the Unix epoch January 1, 1970 00:00 UTC.

We let $\mathsf{clock}()$ denote measurement of the SGX clock from within the enclave, expressed as the current absolute (wall-clock) time.

4.4 A Payment-Free Basic Protocol

For simplicity, we first specify 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 resources and we prove payment-related properties in the paper appendix. For simplicity, we assume a single instance of $\mathsf{prog}_{\text{encl}}$, although our architecture could scale up to multiple enclaves and even server instances. To show messages corresponding to those in Fig. 2, we use the label $(\mathbf{msg}.m_i)$.

The Requester Contract $\mathcal{C}_U$. The requester contract $\mathcal{C}_U$ sends to the TC Contract $\mathcal{C}_{TC}$ a request of the form $(\mathsf{params}, \mathsf{callback})$.

The TC Contract $\mathcal{C}_{TC}$. The TC Contract, as noted above, accepts a datagram request from $\mathcal{C}_U$, assigns a unique id to each request, and records the request. Our Town Crier Relay $\mathcal{R}$ monitors requests received by $\mathcal{C}_{TC}$ and forwards them to an SGX enclave. When $\mathcal{C}_{TC}$ obtains a valid response from $\mathcal{W}_{TC}$, it sends the resulting datagram data to the entry point callback specified by the requesting contract $\mathcal{C}_U$. As explained above, because the response $(\mathbf{msg}.m_2)$ comes from $\mathcal{W}_{TC}$, the blockchain automatically verifies that the response is correctly signed under $\mathsf{prog}_{\text{encl}}$'s key pk_{TC} and $\mathcal{C}_{TC}$ need not verify the signature explicitly. TC does, however, have a subtle security requirement. Specifically, for a given datagram request id, $\mathcal{C}_{TC}$ must verify that $\mathsf{params}' = \mathsf{params}$, where params' is in the digitally signed message produced by $\mathsf{prog}_{\text{encl}}$ 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. 5. Here, Call denotes a call to a contact entry point.

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

Initialize: Counter := 0

Request: On recv $(\mathsf{params}, \mathsf{callback})$ from some $\mathcal{C}_U$:
id:= Counter; Counter := Counter + 1
Record $(\mathsf{id}, \mathsf{params}, \mathsf{callback})$ *// $\mathbf{msg}.m_1$*

Deliver: On recv $(\mathsf{id}, \mathsf{params}, \mathsf{data})$ from $\mathcal{W}_{TC}$:
Retrieve recorded $(\mathsf{id}, \mathsf{params}', \mathsf{callback})$
Assert $\mathsf{params} = \mathsf{params}'$
Call $\mathsf{callback}(\mathsf{data})$ *// $\mathbf{msg}.m_4$*

Figure 5: The Town Crier TC Contract $\mathcal{C}_{TC}$.

The Enclave $\mathsf{prog}_{\text{encl}}$. When initialized through **Initialize**(), $\mathsf{prog}_{\text{encl}}$ ingests the current wall-clock time; by storing this time and setting a clock reference point, it calibrates its absolute clock. It generates an ECDSA keypair $(\mathsf{pk}_{TC}, \mathsf{sk}_{TC})$ (parameterized as in Ethereum), where pk_{TC} is bound to the $\mathsf{prog}_{\text{encl}}$ instance through insertion into attestations.

Upon a call to **Resume**$(\mathsf{id}, \mathsf{params})$, $\mathsf{prog}_{\text{encl}}$ contacts the data source specified by params via HTTPS and checks that the corresponding certificate cert is valid. (We discuss certificate checking in Appendix A.) Then $\mathsf{prog}_{\text{encl}}$ fetches the requested datagram and returns it to $\mathcal{R}$ along with params and id, all digitally signed with sk_{TC}. Fig. 6 shows the protocol for $\mathsf{prog}_{\text{encl}}$.

The Relay $\mathcal{R}$. As noted in Section 3, $\mathcal{R}$ bridges the gap between the Enclave and the blockchain in three ways. (1) It scrapes the blockchain and monitors $\mathcal{C}_{TC}$ for new requests $(\mathsf{id}, \mathsf{params})$. (2) It boots the Enclave with $\mathsf{prog}_{\text{encl}}$.**Initialize**() and calls

Program for Town Crier Enclave ($\mathsf{prog}_{\mathbf{encl}}$)

Initialize (void)
// Subroutine call from $\mathcal{F}_{sgx}$, which attests to
// prog_{encl} and pk_{TC}. See Figure 3.
$(\mathsf{pk}_{TC}, \mathsf{sk}_{TC}) := \Sigma.\mathsf{KeyGen}(1^{\lambda})$
Output pk_{TC}

Resume (id, params)
Parse params as $(\mathsf{url}, \mathsf{spec}, T)$:
Assert $\mathsf{clock}() \geq T.\mathsf{min}$
Contact url via HTTPS, obtaining cert
Verify cert is valid for time $\mathsf{clock}()$
Obtain webpage w from url
Assert $\mathsf{clock}() \leq T.\mathsf{max}$
Parse w to extract data with specification spec
$\sigma := \Sigma.\mathsf{Sign}(\mathsf{sk}_{TC}, (\mathsf{id}, \mathsf{params}, \mathsf{data}))$
Output $((\mathsf{id}, \mathsf{params}, \mathsf{data}), \sigma)$

Figure 6: The Town Crier Enclave $\mathsf{prog}_{\mathsf{encl}}$.

$\mathsf{prog}_{\mathsf{encl}}$.**Resume**(id, params) on incoming requests. (3) it forwards datagram responses from $\mathsf{prog}_{\mathsf{encl}}$ to the blockchain. Recall that it forwards already-signed transacations to the blockchain as $\mathcal{W}_{TC}$. The program for $\mathcal{R}$ is shown in Fig. 7. The function AuthSend inserts a transaction to blockchain ("as $\mathcal{W}_{TC}$" means the transaction is already signed with sk_{TC}). An honest Relay will invoke $\mathsf{prog}_{\mathsf{encl}}$.**Resume** exactly once with the parameters of each valid request and never otherwise.

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

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

Handle(id, params):
Parse params as $(\_, \_, T)$
Wait until $\mathsf{clock}() \geq T.\mathsf{min}$
Send $(\mathtt{resume}, \mathsf{id}, \mathsf{params})$ to $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathsf{encl}}, \mathcal{R}]$
On recv $((\mathsf{id}, \mathsf{params}, \mathsf{data}), \sigma)$ from $\mathcal{F}_{\mathrm{sgx}}[\mathsf{prog}_{\mathsf{encl}}, \mathcal{R}]$:
AuthSend (id, params, data) to $\mathcal{C}_{TC}$ as $\mathcal{W}_{TC}$
// send out $\boldsymbol{msg}.m_3$

Main:
Loop Forever:
Wait for $\mathcal{C}_{TC}$ to records request $(\mathsf{id}, \mathsf{params}, \_)$:
Fork a process of **Handle**(id, params)
End

Figure 7: The Town Crier Relay $\mathcal{R}$.

5 Full Town Crier Protocol

The simplified payment-free protocol in Section 4.4 guarantees authenticity of data for an honest user. In Ethereum, however, computation is not free; recall from Section 2 that Ethereum employs *gas* to fuel contracts. This means that TC needs enough gas to deliver datagrams, so users must pay a fee to reimburse costs. If fees are not properly handled, a malicious relay (or malicious user) could prevent delivery of a datagram or cost an honest requester money for no gain. We discuss Section 5.1 how we address this issue throughout TC's design. We also briefly discuss other enhancements to the basic TC protocol: private and custom datagrams (Section 5.2) and use of replication / voting to achieving robustness against SGX host or data-source compromise (Section 5.3).

5.1 Handling Fees in Ethereum

To address the above concern of attacks on TC fee management, we employ and prove the security of a novel two-currency resource-management system. This system causes requesters to make gas payments up front as ether. It converts this ether to gas so as to prevent a malicious requester from exhausting TC's resources or a malicious TC from stealing an honest user's money. We now give some preliminaries and then explain our system.

Execution model and notation. We take Ethereum's gas model as described in Section 2. We use the notation $\$g$ to denote gas and $\$f$ to denote non-gas currency. In both cases \$ is a type annotation and the letter denotes the numerical amount. For simplicity, our notation assumes that gas and normal currency adopt the same units (allowing us to avoid explicit conversions), so system constants are values without a unit. We use the following identifiers to denote currency and gas amounts.

$\$f$	Currency a requester deposits to refund Town Crier's gas expenditure to deliver a datagram
$\$g_{\mathrm{req}}$ $\$g_{\mathrm{dvr}}$ $\$g_{\mathrm{cncl}}$	`GASLIMIT` when invoking **Request**, **Deliver**, or **Cancel**, respectively
$\$g_{\mathrm{clbk}}$	`GASLIMIT` for callback while executing **Deliver**, set to the max value that can be reimbursed
$\$G_{\mathrm{min}}$	Gas required for **Deliver** excluding callback
$\$G_{\mathrm{max}}$	Maximum gas TC can provide to invoke **Deliver**
$\$G_{\mathrm{cncl}}$	Gas needed to invoke **Cancel**
$\$G_{\emptyset}$	Gas needed for **Deliver** on a canceled request

$\$G_{\mathrm{min}}$, $\$G_{\mathrm{max}}$, $\$G_{\mathrm{cncl}}$, and $\$G_{\emptyset}$ are system constants, $\$f$ is chosen by the requester (and may be malicious if the requester is dishonest), and $\$g_{\mathrm{dvr}}$ is chosen by the TC Enclave when calling **Deliver**. Though $\$g_{\mathrm{req}}$ and $\$g_{\mathrm{cncl}}$ are set by the requester, we need not worry about the values.

If they are too small, Ethereum will abort the transaction and there will not be a request or cancellation.

Adversarial cases. During the creation and fulfillment of any request, there are two untrusted parties: The requester contract / user $\mathcal{C}_U$ and the Relay. In the TC implementation, we thus consider three cases, here giving the desired properties of each:

- *Honest requester and Relay.* The requester must receive a valid authenticated response from TC.
- *Malicious requester and honest Relay.* TC must still be able to respond to requests from other (honest) users. Thus we must prevent a malicious user from interfering directly with other requests (which the payment-free protocol already does) or exhausting the financial resources of TC.
- *Honest requester and malicious Relay.* The requester cannot receive invalid data (which is also assured by the payment-free protocol) and the requester should not have to pay for computation that is not executed.

We formalize these properties in Section 6 and prove that our protocol provides these guarantees. We intentionally ignore the case where both the requester and the Relay are dishonest. If the requester is dishonest we need to not protect her request, and if the Relay is dishonest we cannot protect the TC system.

Town Crier protocol with fees. Our basic Town Crier system implements a policy where the requester pays for all gas needed and Town Crier effectively pays nothing. We now describe how this can be realized by modifying the payment-free protocol described in Section 4.4.

- *Initialization.* We assume that TC deposits at least $\$G_{max}$ into the wallet $\mathcal{W}_{TC}$.
- *Town Crier blockchain contract.* Figure 8 specifies the TC blockchain contract $\mathcal{C}_{TC}$, with fees. Since $\mathcal{W}_{TC}$ must invoke **Deliver**, TC pays the gas cost. It sets the `GASLIMIT` $\$g_{dvr} := \$G_{max}$. To ensure that the gas spent does not exceed the reimbursement available ($\$f), $\mathcal{C}_{TC}$ sets the `GASLIMIT` $\$g_{clbk}$ for the sub-call to callback to the remaining reimbursement: $\$f - \$G_{min}$.
- *Town Crier Relay.* The Relay behavior does not change with the presence of fees. It still monitors the blockchain and whenever the contract $\mathcal{C}_{TC}$ stores a new request $(\mathsf{id}, \mathsf{params}, \_, \_, \_)$, it invokes $\mathsf{prog}_{\mathsf{encl}}$.**Resume**$(\mathsf{id}, \mathsf{params})$.
- *Town Crier Enclave.* We make the following small modification to the fee-free protocol. Instead of signing the tuple $(\mathsf{id}, \mathsf{params}, \mathsf{data})$ at the end of its execution, the enclave now signs the tuple $(\mathsf{id}, \mathsf{params}, \mathsf{data}, \$g_{dvr})$ where $\$g_{dvr} = \$G_{max}$.

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

Initialize: Counter $:= 0$

Request: On recv $(\mathsf{params}, \mathsf{callback}, \$f, \$g_{req})$ from some $\mathcal{C}_U$:
Assert $\$G_{min} \le \$f \le \$G_{max}$
$\mathsf{id} :=$ Counter; Counter $:=$ Counter $+ 1$
Store $(\mathsf{id}, \mathsf{params}, \mathsf{callback}, \$f, \mathcal{C}_U)$
// $\$f held by contract

Deliver: On recv $(\mathsf{id}, \mathsf{params}, \mathsf{data}, \$g_{dvr})$ from $\mathcal{W}_{TC}$:
(1) If isCanceled[id] and not isDelivered[id]
Set isDelivered[id]
(2) Send $\$G_{\emptyset}$ to $\mathcal{W}_{TC}$
Return
Retrieve stored $(\mathsf{id}, \mathsf{params}', \mathsf{callback}, \$f, \_)$
// abort if not found
Assert params $=$ params$'$ and $\$f \le \$g_{dvr}$
and isDelivered[id] not set
Set isDelievered[id]
(3) Send $\$f$ to $\mathcal{W}_{TC}$
Set $\$g_{clbk} := \$f - \$G_{min}$
(4) Call callback(data) with $\$g_{clbk}$ max gas

Cancel: On recv $(\mathsf{id}, \$g_{cncl})$ from $\mathcal{C}_U$:
Retrieve stored $(\mathsf{id}, \_, \_, \$f, \mathcal{C}'_U)$
// abort if not found
Assert $\mathcal{C}_U = \mathcal{C}'_U$ and $\$f \ge \$G_{\emptyset}$
and isDelivered[id] not set
and isCanceled[id] not set
Set isCanceled[id]
(5) Send $(\$f - \$G_{\emptyset})$ to $\mathcal{C}_U$ *// hold* $\$G_{\emptyset}$

Figure 8: Town Crier contract $\mathcal{C}_{TC}$ reflecting fees. The last argument of each entry point is the `GASLIMIT` provided. An honest requester sets $\$f$ to be the gas required to execute **Deliver** including callback. Town Crier sets $\$g_{dvr} := \$G_{max}$, but lowers the gas limit for callback to ensure that no more than $\$f$ is spent.

Program for Town Crier Enclave $(\mathsf{prog}_{\mathbf{encl}})$

Initialize (void) [Same as Figure 6]

Resume $(\mathsf{id}, \mathsf{params})$
[Same as Figure 6 except the last two lines:]
Set $\$g_{dvr} := \$G_{max}$
$\sigma := \Sigma.\mathsf{Sign}(\mathsf{sk}_{TC}, (\mathsf{id}, \mathsf{params}, \mathsf{data}, \$g_{dvr}))$
Output $((\mathsf{id}, \mathsf{params}, \mathsf{data}, \$g_{dvr}), \sigma)$

Figure 9: The Town Crier Enclave $\mathsf{prog}_{\mathsf{encl}}$.

- *Requester.* An honest requester follows the protocol in Fig. 4 to verify the SGX attestation. Then she prepares params and callback, sets $\$g_{req}$ to the gas cost of **Request** with params, and sets $\$f$ to $\$G_{min}$ plus the cost of executing callback.

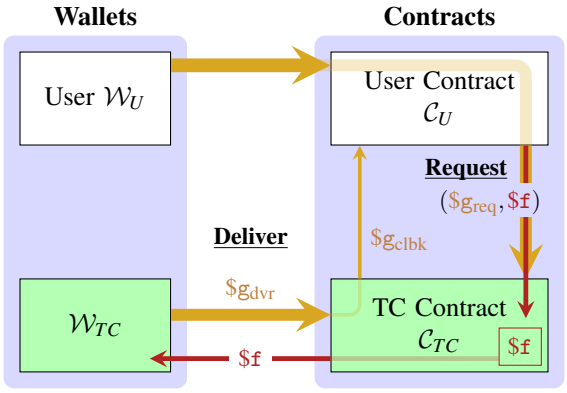

Figure 10: **Money Flow for a Delivered Request.** Red arrows denote flow of money and brown arrows denote gas limits for functions. The thickness of the line indicates the quantity of the resource. The $\$g_{clbk}$ arrow is thin because the value is limited to $\$f - \$G_{min}$.

Finally, she invokes **Request**(params, callback, $f) with GASLIMIT g_{req}.

If callback is not executed, the requester will invoke **Cancel** with argument id and gas limit $\$G_{cncl}$ to receive a partial refund. The honest requester will never invoke **Cancel** more than once for the same request and will never invoke **Cancel** with an id that corresponds to a different user's request.

5.2 Private and Custom Datagrams

In addition to ordinary datagrams, TC supports *private datagrams*, which are requests where params includes ciphertexts under pk_{TC}. Private datagrams can thus enable confidentiality-preserving applications despite the public readability of the blockchain. *Custom datagrams*, also supported by TC, allow a contract to specify a particular web-scraping target, potentially involving multiple interactions, and thus greatly expand the range of possible relying contracts for TC. We do not treat them in our security proofs, but give examples of both datagram types in Section 7.

5.3 Enhanced Robustness via Replication

Our basic security model for TC assumes the ideal isolation model for SGX described above as well as client trust in data sources. Given various concerns about SGX security [20, 36] and the possible fallibility of data sources, we examine two important ways TC can support hedging. To protect against the compromise of a single SGX instance, contracts may request datagrams from multiple SGX instances and implement majority voting among the responses. This hedge requires increased gas expenditure for additional requests and storage of returned data. Similarly, TC can hedge against the compromise of a data source by scraping multiple sources for the same data and selecting the majority response. We demonstrate both of these mechanisms in our example financial derivative application in Section 8.4. (A potential optimization is mentioned in Appendix E.)

5.4 Implementation Details

We implemented a full version of the TC protocol in a complete, end-to-end system using Intel SGX and Ethereum. Due to space constraints, we defer discussion of implementation details and other practical considerations to Appendix A.

6 Security Analysis

Proofs of theorems in this section appear in Appendix C.

Authenticity. Intuitively, authenticity means that an adversary (including a corrupt user, Relay, or collusion thereof) cannot convince $\mathcal{C}_{TC}$ to accept a datagram that differs from the expected content obtained by crawling the specified url at the specified time. In our formal definition, we assume that the user and $\mathcal{C}_{TC}$ behave honestly. Recall that the user must verify upfront the attestation σ_{att} that vouches for the enclave's public key pk_{TC}.

Definition 1 (Authenticity of Data Feed)**.** *We say that the TC protocol satisfies* Authenticity of Data Feed *if, for any polynomial-time adversary $\mathcal{A}$ that can interact arbitrarily with $\mathcal{F}_{sgx}$, $\mathcal{A}$ cannot cause an honest verifier to accept $(\mathsf{pk}_{TC}, \sigma_{att}, \mathsf{params} := (\mathsf{url}, \mathsf{pk}_{url}, T), \mathsf{data}, \sigma)$ where* data *is not the contents of* url *with the public key* pk_{url} *at time T ($\mathsf{prog}_{encl}(\mathsf{params})$ in our model). More formally, for any probabilistic polynomial-time adversary $\mathcal{A}$,*

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

for security parameter λ.

Theorem 1 (Authenticity)**.** *Assume that Σ_{sgx} and Σ are secure signature schemes. Then, the full TC protocol achieves authenticity of data feed under Definition 1.*[3]

Fee Safety. Our protocol in Section 5.1 ensures that an honest Town Crier will not run out of money and that an honest requester will not pay excessive fees.

[3] Recall that we model SGX's group signature as a regular signature scheme under a manufacturer public key pk_{sgx} using the model in [32].

Theorem 2 (Gas neutrality for Town Crier)**.** *If the TC Relay is honest, Town Crier's wallet $\mathcal{W}_{TC}$ will have at least $\$\mathtt{G}_{max}$ remaining after each* **Deliver** *call.*

An honest user should only have to pay for computation that is executed honestly on her behalf. If a valid datagram is delivered, this is a constant value plus the cost of executing callback. Otherwise the requester should be able to recover the cost of executing **Deliver**. For Theorem 2 to hold, $\mathcal{C}_{TC}$ must retain a small fee on cancellation, but we allow the user to recover all but this small constant amount. We now formalize this intuition.

Theorem 3 (Fair Expenditure for Honest Requester)**.** *For any* params *and* callback*, let $\$\mathtt{G}_{req}$ and $\$\mathtt{F}$ be the respective values chosen by an honest requester for $\$\mathtt{g}_{req}$ and $\$\mathtt{f}$ when submitting the request* $(\mathsf{params}, \mathsf{callback}, \$\mathtt{f}, \$\mathtt{g}_{req})$*. For any such request submitted by an honest user $\mathcal{C}_U$, one of the following holds:*

- callback *is invoked with a valid datagram matching the request parameters* params*, and the requester spends at most $\$\mathtt{G}_{req} + \$\mathtt{G}_{cncl} + \$\mathtt{F}$;*
- *The requester spends at most $\$\mathtt{G}_{req} + \$\mathtt{G}_{cncl} + \$\mathtt{G}_{\emptyset}$.*

Other security concerns. In Section 5.3, we addressed concerns about attacks outside the SGX isolation model embraced in the basic TC protocol. A threat we do not address in TC is the risk of traffic analysis by a network adversary or compromised Relay against confidential applications (e.g. with private datagrams), although we briefly discuss the issue in Section 7. We also note that while TC assumes the correctness of data sources, if a scraping failure occurs, TC delivers an empty datagram, enabling relying contracts to fail gracefully.

7 Applications: Requesting Contracts

We built and implemented several showcase applications which we plan to launch together with Town Crier. We give a description of these applications in this section, and show experimental results in Section 8. We refer the reader to Appendix D for more details on these applications, as well as $\mathcal{C}_U$ code samples that demonstrate how to use Town Crier's service.

Financial derivative (CashSettledPut)**.** Financial derivatives are among the most commonly cited smart contract applications, and exemplify the need for a data feed on financial instruments. We implemented an example contract CashSettledPut for a *cash-settled put option*. This is an agreement for one party to buy an asset from the other at an agreed upon price on or before a particular date. It is "cash-settled" in that the sale is implicit, i.e. no asset changes hands, only cash reflecting the asset's value.

Flight insurance (FlightIns)**.** Flight insurance indemnifies a purchaser should her flight be delayed or canceled. We have implemented a simple flight insurance contract called FlightIns. Our implementation showcases TC's *private-datagram* feature to address an obvious concern: Customers may not wish to reveal their travel plans publicly on the blockchain. Roughly speaking, a customer submits to $\mathcal{C}_{TC}$ a request $\mathsf{Enc}_{\mathsf{pk}_{TC}}(\mathsf{req})$ encrypted under Town Crier enclave's public key pk_{TC}. The enclave decrypts req and checks that it is well-formed (e.g., submitted sufficiently long before the flight time). The enclave will then fetch the flight information from a target website at a specified later time, and send to $\mathcal{C}_{TC}$ a datagram indicating whether the flight is canceled. Finally, to avoid leaking information through timing (e.g., when the flight information website is accessed or datagram sent), random delays are introduced.

Steam Marketplace (SteamTrade)**.** Authenticated data feeds and smart contracts can enable fair exchange of digital goods between Internet users who do not have pre-established trust. We have developed an example application supporting fair trade of virtual items for Steam [5], an online gaming platform that supports thousands of games and maintains its own marketplace, where users can trade, buy, and sell games and other virtual items. We implemented a contract for the sale of games and items for ether that showcases TC's support for *custom datagrams* through the use of Steam's access-controlled API. In our implementation, the seller sends $\mathsf{Enc}_{\mathsf{pk}_{TC}}(\text{account credentials}, \mathsf{req})$ to $\mathcal{C}_{TC}$, such that TC's enclave can log in as the seller and determine from the web-page whether the virtual item has been shipped.

8 Experiments

We evaluated the performance of TC on a Dell Inspiron 13-7359 laptop with an Intel i7-6500U CPU and 8.00GB memory, one of the few SGX-enabled systems commercially available at the time of writing. We show that on this single host—not even a server, but a consumer device—our implementation of TC can easily process transactions at the peak global rate of Bitcoin, currently the most heavily loaded decentralized blockchain.

8.1 Enclave Response Time

We first measured the enclave response time for handling a TC request, defined as the difference in time between (1) the Relay sending a request to the enclave and (2) the Relay receiving a response back from the enclave.

Table 1 summarizes the total enclave response time as well as its breakdown over 500 runs. For the three applications we implemented, the enclave response time ranges from **180 ms** to **599 ms**. The response time is

	CashSettledPut					FlightIns					SteamTrade				
	mean	%	t_{max}	t_{min}	σ_t	**mean**	%	t_{max}	t_{min}	σ_t	**mean**	%	t_{max}	t_{min}	σ_t
Ctx. switch	1.00	0.6	3.12	0.25	0.31	1.23	0.24	2.94	0.17	0.32	1.17	0.20	3.25	0.36	0.35
Web scraper	157	87.2	258	135	18	482	95.4	600	418	31	576	96.2	765	489	52
Sign	20.2	11.2	26.6	18.7	1.52	20.5	4.0	25.3	18.9	1.4	20.3	3.4	24.8	18.8	1.28
Serialization	0.40	0.2	0.84	0.24	0.08	0.38	0.08	0.67	0.20	0.08	0.39	0.07	0.65	0.24	0.09
Total	180	100	284	158	18	505	100	623	439	31	599	100	787	510	52

Table 1: Enclave response time t, with profiling breakdown. All times are in **milliseconds**. We executed 500 experimental runs, and report the statistics including the average (**mean**), proportion (%), maximum (t_{max}), minimum (t_{min}) and standard deviation (σ_t). Note that **Total** is the end-to-end response time as defined in Section 8.1. Times may not sum to this total due to minor unprofiled overhead.

clearly dominated by the web scraper time, i.e., the time it takes to fetch the requested information from a website. Among the three applications evaluated, SteamTrade has the longest web scraper time, as it interacts with the target website over multiple roundtrips to fetch the desired datagram.

8.2 Transaction Throughput

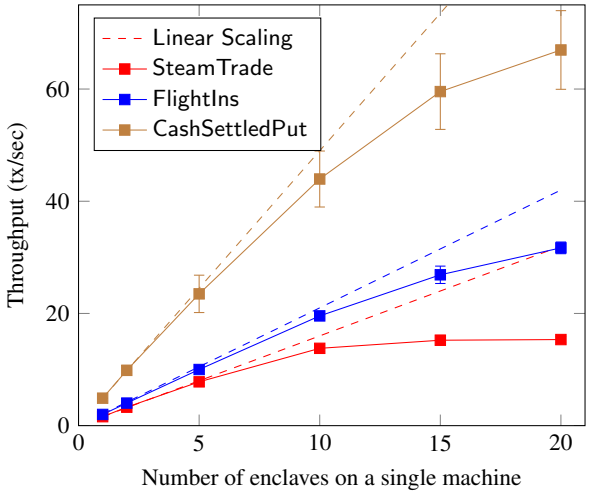

Figure 11: Throughput on a single SGX machine. The x-axis is the number of concurrent enclaves and the y-axis is the number of tx / sec. Dashed lines indicate the ideal scaling for each application, and error bars, the standard deviation. We ran 20 rounds of experiments (each round processing 1000 transactions in parallel).

We performed a sequence of experiments measuring the transaction throughput while scaling up the number of concurrently running enclaves on our single SGX-enabled host from 1 to 20. 20 TC enclaves is the maximum possible given the enclave memory constraints on the specific machine model we used. Figure 11 shows that for the three applications evaluated, **a single SGX machine can handle 15 to 65 tx/sec**.

Several significant data points show how effectively TC can serve the needs of today's blockchains for authenticated data: Ethereum currently handles < 1 tx/sec on average. Bitcoin today handles slightly more than 3 tx/sec, and its maximum throughput (with full block utilization) is roughly 7 tx/sec. We know of no measurement study of the throughput bound of the Ethereum peer-to-peer network. It has been shown that without a protocol redesign, however, the current Bitcoin network cannot scale via reparametrization beyond 27 tx/sec [21]. Thus, with one or at most a few hosts, TC can easily meet the data feed demands of even future decentralized blockchains.

8.3 Gas Costs

Currently 1 gas costs 5×10^{-8} ether, so at the exchange rate of $5 for 1 ether, $1 buys 4 million gas. Here we provide costs for the components of our implementation.

Table 2 shows gas costs for calling TC in our example applications; these costs are callback-independent to reflect datagram costs only, not application costs. We see an effective gas cost for TC of roughly 4 to 5 cents.

The callback-independent portion of **Deliver** costs about 35,000 gas (0.9¢), so this is the value of $\$G_{min}$. We set $\$G_{max} = 3{,}100{,}000$ gas (77.5¢), as this is approximately Ethereum's maximum `GASLIMIT`. The cost for executing **Request** is approximately 120,000 gas (3¢) of fixed cost, plus 2500 gas (0.06¢) for every 32 bytes of request parameters. The cost to execute **Cancel** is 62500 gas (1.55¢) including the gas cost $\$G_{cncl}$ and the refund $\$G_{\emptyset}$ paid to TC should **Deliver** be called after **Cancel**.

8.4 Component-Compromise Resilience

For the CashSettledPut application, we implemented and evaluated two modes of majority voting (as in Section 5.3):

- 2-out-of-3 majority voting within the enclave, provid-

	CashSettledPut	FlightIns	SteamTrade†
Deliver without Cancel	3.95¢	4.00¢	4.30¢
Cancel arrived after Deliver	4.90¢	4.95¢	5.25¢
Cancel without Deliver	4.65¢	4.70¢	5.00¢

Table 2: **Callback-independent** portion of gas expenditure in USD. The difference between applications is due to the differing lengths of the input parameters. The first two rows would also have to pay for callback, but we do not include that cost as it would exist even if data acquisition were free.
† These numbers are for 1 item. Each additional item costs an additional 0.06¢.

ing robustness against data-source compromise. In our experiments the enclave performed simple sequential scraping of current stock prices from three different data sources: Bloomberg, Google Finance and Yahoo Finance. The enclave response time is roughly 1743(109) ms in this case (*c.f.* 1058(88), 423(34) and 262(12) ms for each respective data source). Finally, there is no change in gas cost, as voting is done inside the SGX enclave. In the future, we will investigate parallelization of SGX's thread mechanism, with careful consideration of the security implications.

- 2-out-of-3 majority voting within the requester contract, which provides robustness against SGX compromise. We ran three instances of SGX enclaves, all scraping the same data source. The main change in this scenario is that the gas cost would increase by a factor of 3 plus an additional 1.95¢. So CashSettledPut would cost 13.8¢ for Deliver without Cancel. The extra 1.95¢ is a storage cost: The requester contract must store votes until a winner is known.

8.5 Offline Measurements

Recall that a one-time setup operation is required for an enclave, and involves attestation generation. We report mean run times (with the standard deviation in parentheses) for these experiments, which involved 100 trials. Setting up the TC Enclave takes 49.5 (7.2) ms, and attestation generation takes 61.9 (10.7) ms, including 7.65 (0.97) ms for report generation, and 54.9 (10.3) ms for quote generation.

Recall also that since clock() yields only relative time in SGX, TC's absolute clock is calibrated through an externally furnished wall-clock timestamp. A user can verify the correctness of the Enclave absolute clock by requesting a digitally signed timestamp. This verification procedure is, of course, accurate only to within its end-to-end latency. Our experiments show that the time between Relay transmission of a clock calibration request to the enclave and receipt of a response is 11.4 (1.9) ms of which 10.5 (1.9) ms is to sign the timestamp. To this must be added the wide-area network roundtrip latency, at most typically a few hundred milliseconds.

9 Related Work

Several data feeds are deployed today for smart contract systems such as Ethereum. Examples include PriceFeed [4] and Oraclize.it [8]. The latter achieves distributed trust by using a second service called TLSnotary [7], which digitally signs webpages. These services, however, ultimately rely on the reputations of their (small) providers to ensure data authenticity. To address this problem, systems such as SchellingCoin [14] and Augur [3] rely on mechanisms such as prediction markets to decentralize trust. Prediction markets often rely on human input, though, constraining their scope. They have not yet seen widespread use in cryptocurrencies.

Despite an active developer community, research community exploration of smart contracts has been very limited to date. Work includes off-chain contract execution for confidentiality [26], and, more tangentially, exploration of e.g., randomness sources in [12]. The only exploration relating to data feeds of which we're aware is discussion of their use for (criminal) applications in [25].

SGX is similarly in its infancy. While a Windows SDK [24] and programming manual [23] have just been released, a number of pre-release papers have already explored SGX, e.g., [9, 27, 29, 30, 36]. Researchers have demonstrated applications including enclave execution of legacy (non-SGX) code [11] and use of SGX in a distributed setting for map-reduce computations [30]. Several works have exposed shortcomings of the security model for SGX [20, 31, 32], including side-channel attacks and other attacks against enclave state.

10 Conclusion

We have introduced Town Crier (TC), an authenticated data feed for smart contracts specifically designed to support Ethereum. Thanks to its use of Intel's new SGX trusted hardware, TC serves datagrams with a high degree of trustworthiness. We prove in a formal model capturing SGX and blockchain behavior that TC serves only data from authentic sources. We also prove that its novel gas-management system achieves sustainable gas use if the unprotected server code (outside SGX) in TC

behaves honestly and minimizes gas losses should the code behave maliciously. In experiments involving end-to-end use of the complete system with the Ethereum blockchain, we demonstrated TC's practicality, cost effectiveness, and flexibility for three example smart contract applications. We believe that TC offers a powerful, practical means to address the lack of trustworthy data feeds hampering Ethereum evolution today and will support a rich range of applications. Pending deployment of the Intel attestation verification service in the near future, we will make TC freely available as a public service.

References

[1] Private communication with blockstream.

[2] `http://coinmarketcap.com/currencies/ethereum`.

[3] Augur. `http://www.augur.net/`.

[4] PriceFeed smart contract. Referenced Feb. 2016 at http://feed.ether.camp/.

[5] Steam online gaming platform. `http://store.steampowered.com/`.

[6] Vitalik Buterin (personal communication), Jan. 2016.

[7] TLSnotary - a mechanism for independently audited https sessions. https://tlsnotary.org/TLSNotary.pdf, 10 Sept. 2014.

[8] Oraclize: "The provably honest oracle service". www.oraclize.it, Referenced Feb. 2016.

[9] ANATI, I., GUERON, S., AND JOHNSON, S. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[10] ARM LIMITED. mbed TLS (formerly known as PolarSSL). `https://tls.mbed.org/`.

[11] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI* (2014).

[12] BONNEAU, J., CLARK, J., AND GOLDFEDER, S. On bitcoin as a public randomness source. *URL https://eprint. iacr. org/2015/1015. pdf* (2015).

[13] BRICKELL, E., AND LI, J. Enhanced privacy id from bilinear pairing. *IACR Cryptology ePrint Archive 2009* (2009), 95.

[14] BUTERIN, V. Schellingcoin: A minimal-trust universal data feed. `https://blog.ethereum.org/2014/03/28/schellingcoin-a-minimal-trust-universal-data-feed/`.

[15] BUTERIN, V. Ethereum: A next-generation smart contract and decentralized application platform.

[16] CANETTI, R. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS* (2001).

[17] CANETTI, R., DODIS, Y., PASS, R., AND WALFISH, S. Universally composable security with global setup. In *Theory of Cryptography*. Springer, 2007, pp. 61–85.

[18] CANETTI, R., AND RABIN, T. Universal composition with joint state. In *CRYPTO* (2003).

[19] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 191–206.

[20] COSTAN, V., AND DEVADAS, S. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. `http://eprint.iacr.org/`.

[21] CROMAN, K., DECKER, C., EYAL, I., GENCER, A. E., JUELS, A., KOSBA, A., MILLER, A., SAXENA, P., SHI, E., SIRER, E. G., SONG, D., AND WATTENHOFER, R. On scaling decentralized blockchains (a position paper). In *Bitcoin Workshop* (2016).

[22] ETHEREUM. go-ethereum. `https://github.com/ethereum/go-ethereum`.

[23] INTEL CORP. *Intel(R) Software Guard Extensions Programming Reference*, 329298-002us ed., 2014.

[24] INTEL CORP. Intel(R) Software Guard Extensions (Intel(R) SGX) SDK. `https://software.intel.com/en-us/sgx-sdk`, 2015.

[25] JUELS, A., KOSBA, A., AND SHI, E. The Ring of Gyges: Investigating the future of criminal smart contracts. Online manuscript, 2015.

[26] KOSBA, A., MILLER, A., SHI, E., WEN, Z., AND PAPAMANTHOU, C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy* (2016).

[27] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[28] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system.

[29] PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, New York, USA, 2013), ACM Press, pp. 1–1.

[30] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy data analytics in the cloud. In *IEEE S& P* (2015).

[31] SHI, E. Trusted hardware: Life, the composable university, and everything. Talk at the DIMACS Workshop on Cryptography and Big Data, 2015.

[32] SHI, E., ZHANG, F., PASS, R., DEVADAS, S., SONG, D., AND LIU, C. Trusted hardware: Life, the composable universe, and everything. Manuscript, 2015.

[33] SZABO, N. Smart contracts. `http://szabo.best.vwh.net/smart.contracts.html`, 1994.

[34] TORPEY, K. The conceptual godfather of augur thinks the project will fail. CoinGecko, 5 Aug. 2015.

[35] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014).

[36] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on* (May 2015), pp. 640–656.

A TC Implementation Details

We now present further, system-level details on the TC contract $\mathcal{C}_{TC}$ and the two parts of the TC server, the Enclave and Relay.

A.1 TC Contract

We implement $\mathcal{C}_{TC}$ with fees as described in Section 5.1 in Solidity, a high-level language with JavaScript-like syntax which compiles to Ethereum Virtual Machine bytecode—the language Ethereum contracts use.

In order to handle the most general type of requests—including encrypted parameters—the $\mathcal{C}_{TC}$ implementation requires two parameter fields: an integer specifying the type of request (e.g. flight status) and a byte array of user-specified size. This byte array is parsed and interpreted inside the Enclave, but is treated as an opaque byte array by $\mathcal{C}_{TC}$. For convenience, we include the timestamp of the current block as an implicit parameter.

To guard against the Relay tampering with request parameters, the $\mathcal{C}_{TC}$ protocol includes params as an argument to **Deliver** which validates against stored values. To reduce this cost for large arrays, we store and verify SHA3-256(requestType||timestamp||paramArray). The Relay scrapes the raw values for the Enclave which computes the hash and includes it as an argument to **Deliver**.

As we mentioned in Section 4.2, to allow for improved efficiency in client contracts, **Request** returns id and **Deliver** includes id along with data as arguments to callback. This allows client contracts to make multiple requests in parallel and differentiate the responses, so it is no longer necessary to create a unique client contract for every request to $\mathcal{C}_{TC}$.

A.2 TC Server

Using the recently released Intel SGX SDK [24], we implemented the TC Server as an SGX-enabled application in C++. In the programming model supported by the SGX SDK, the body of an SGX-enabled application runs as an ordinary user-space application, while a relatively small piece of security-sensitive code runs in the isolated environment of the SGX enclave.

The enclave portion of an SGX-enabled application may be viewed as a shared library exposing an API in the form of *ecalls* [24] to be invoked by the untrusted application. Invocation of an ecall transfers control to the enclave; the enclave code runs until it either terminates and explicitly releases control, or some special event (e.g., exception) happens [23]. Again, as we assume SGX provides ideal isolation, the untrusted application cannot observe or alter the execution of ecalls.

Enclave programs can make *ocalls* [24] to invoke functions defined outside of the enclave. An ocall triggers an exit from the enclave; control is returned once the ocall completes. As ocalls execute outside the enclave, they must be treated by enclave code as untrusted.

For TC, we recall that Fig. 6 shows the Enclave code

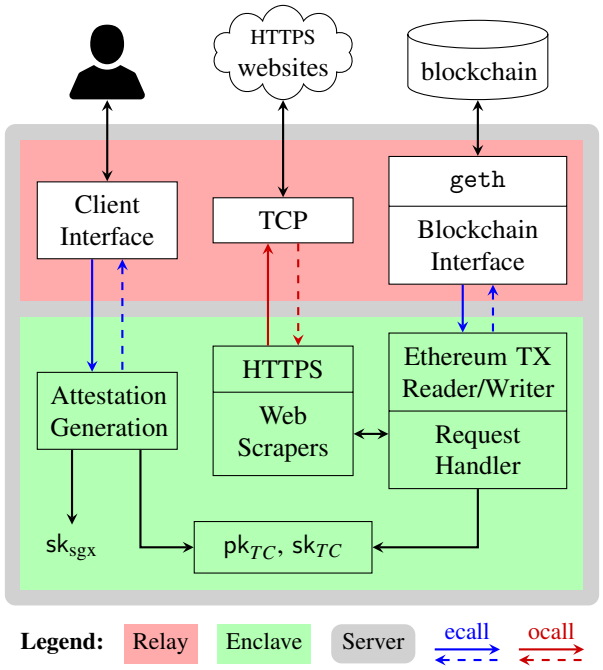

Figure 12: Components of TC Server

$\text{prog}_{\text{encl}}$. Fig. 7 specifies the operation of the Relay, the untrusted code in TC, which we emphasize again provides essentially only network functionality. We now give details on the services in the Enclave and the Relay and describe their interaction, as summarized in Fig. 12.

The Enclave. There are three components to the enclave code $\text{prog}_{\text{encl}}$: an HTTPS service, Web Scrapers, which interact with data sources, and a Request Handler, which services datagram requests.

HTTPS Service. We recall that the enclave does not have direct access to host network functionality. TC thus partitions HTTPS into a trusted layer, consisting of HTTP and TLS code, and an untrusted layer that provides low-layer network service, specifically TCP. This arrangement allows the enclave to establish a secure channel with a web server; the enclave itself performs the TLS handshake with a target server and performs all cryptographic operations internally, while the untrusted process acts as a network interface only. We ported a TLS library (mbedTLS [10]) and HTTP code into the SGX environment. We minimized the HTTP code to meet the web-scraping requirements of TC while keeping the TCB small. To verify certificates presented by remote servers, we hardcoded a collection of root CA certificates into the enclave code; in the first version of TC, the root CAs are identical to those in Chrome. By using its internal, trusted wall-clock time, it is possible to verify that a certificate has not expired. (We briefly discuss revocation

in Appendix E.)

Web Scrapers. We implemented scrapers for our examples in Section 7 in an ad hoc manner for our initial implementation of TC. We defer more principled, robust approaches to future work.

Request Handler. The Request Handler has two jobs.

1. It ingests a datagram request in Ethereum's serialization format, parses it, and decrypts it (if it is a private-datagram request).
2. It generates an Ethereum transaction containing the requested datagram (and parameter hash), serializes it as a blockchain transaction, signs it using sk_{TC}, and furnishes it to the Relay.

We implemented the Ethereum ABI and RLP which, respectively, specify the serialization of arguments and transactions in Ethereum.

Attestation Generation. Recall in Section 2 we mentioned that an *attestation* is an *report* digitally signed by the Intel-provided Quoting Enclave (QE). Therefore two phases are involved in generating att. First, the Enclave calls `sgx_create_report` to generate a report with QE as the target enclave. Then the Relay forwards the report to QE and calls `sgx_get_quote` to get a signed version of the report, namely an attestation.

The Relay. The Relay encompasses three components: A Client Interface, which serves attestations and timestamps, OS services, including networking and time services, and a Blockchain Interface.

Client Interface. As described in Section 3, a client starts using TC by requesting and verifying an attestation att and checking the correctness of the clock in the TC enclave using a fresh timestamp. The Client Interface caches att upon initialization of $\mathsf{prog}_{\text{encl}}$. When it receives a web request from a client for an attestation, it issues an ecall to the enclave to obtain a Unix timestamp signed using sk_{TC}, which it returns to the client along with att. The client verify att using the Intel Attestation Service (IAS) and then verify the timestamp using pk_{TC} and check it using any trustworthy time service.

OS services. The Enclave relies on the Relay to access networking and wall-clock time (used for initialization) provided by the OS and implemented as ocalls.

Blockchain Interface. The Relay's Blockchain Interface monitors the blockchain for incoming requests and places transactions on the blockchain in order to deliver datagrams. The Blockchain Interface incorporates an official Ethereum client, Geth [22]. This Geth client can be configured with a JSON RPC server. The Relay communicates with the blockchain indirectly via RPC calls to this server. For example, to insert a signed transaction, the Relay simply calls `eth_sendRawTransaction` with the byte array of the serialized transaction. We emphasize that, as the enclave holds sk_{TC}, transactions are signed within the enclave.

B More Details on Formal Modeling

B.1 SGX Formal Modeling

As mentioned earlier, we adopt the UC model of SGX proposed by Shi et al. [32] In particular, their abstraction captures a subset of the features of Intel SGX. The main idea behind the UC modeling by Shi et al. [32] is to think of SGX as a trusted third party defined by a global functionality $\mathcal{F}_{\text{sgx}}$ (see Figure 3 of Section 4.3).

Modeling choices. For simplicity, the $\mathcal{F}_{\text{sgx}}$ model currently does not capture the issue of revocation. In this case, as Shi et al. point out, we can model SGX's group signature simply as a regular signature scheme Σ_{sgx}, whose public and secret keys are called "manufacturer keys" and denoted pk_{sgx} and sk_{sgx} (i.e., think of always signing with the 0-th key of the group signature scheme). We adopt this notational choice from [32] for simplicity. Later when we need to take revocation into account, it is always possible to replace this signature scheme with a group signature scheme in the modeling.

The $\mathcal{F}_{\text{sgx}}(\Sigma_{\text{sgx}})$ functionality described by Shi et al. [32] is a global functionality shared by all protocols, parametrized by a signature scheme Σ_{sgx}. This global $\mathcal{F}_{\text{sgx}}$ is meant to capture all SGX machines available in the world, and keeps track of multiple execution contexts for multiple enclave programs, happening on different SGX machines in the world. For convenience, this paper adopts a new notation $\mathcal{F}_{\text{sgx}}(\Sigma_{\text{sgx}})[\mathsf{prog}_{\text{encl}}, \mathcal{R}]$ to denote one specific execution context of the global $\mathcal{F}_{\text{sgx}}$ functionality where the enclave program in question is $\mathsf{prog}_{\text{encl}}$, and the specific SGX instance is attached to a physical machine $\mathcal{R}$. (As the *Relay* in TC describes all functionality outside the enclave, we use $\mathcal{R}$ for convenience also to denote the physical host.) This specific context $\mathcal{F}_{\text{sgx}}(\Sigma_{\text{sgx}})[\mathsf{prog}_{\text{encl}}, \mathcal{R}]$ ignores all parties' inputs except those coming from $\mathcal{R}$. We often omit writing (Σ_{sgx}) without risk of ambiguity.

Operations. $\mathcal{F}_{\text{sgx}}$ captures the following features:

- *Initialize.* Initialization is run only once. Upon receiving `init`, $\mathcal{F}_{\text{sgx}}$ runs the initialization part of the enclave program denoted $\mathtt{outp} := \mathsf{prog}_{\text{encl}}.\textbf{Initialize}()$. Then, $\mathcal{F}_{\text{sgx}}$ attests to the code of the enclave program $\mathsf{prog}_{\text{encl}}$ as well as outp. The resulting attestation is denoted σ_{att}.
- *Resume.* When `resume` is received, $\mathcal{F}_{\text{sgx}}$ calls

$\mathsf{prog}_{\text{encl}}$.**Resume** on the input parameters denoted params. $\mathcal{F}_{\text{sgx}}$ outputs whatever $\mathsf{prog}_{\text{encl}}$.**Resume** outputs. $\mathcal{F}_{\text{sgx}}$ is stateful, i.e., allowed to carry state between `init` and multiple `resume` invocations.

Finally, we remark that this formal model by Shi et al. is speculative, since we know of no formal proof that Intel's SGX does securely realize this abstraction (or realize any useful formal abstraction at all for that matter)—in fact, available public documentation of SGX does not provide sufficient information for making such formal proofs. As such, the formal model in [32] appears to be the best available tool for us to formally reason about security for SGX-based protocols. Shi et al. leave it as an open question to design secure processors with clear formal specifications, such that they can be used in the design of larger protocols/systems supporting formal reasoning of security. We refer the readers to [32] for a more detailed description of the UC modeling of Intel SGX.

B.2 Blockchain Formal Modeling

Our protocol notation adopts the formal blockchain framework recently proposed by Kosba et al. [26]. In addition to UC modeling of blockchain-based protocols, Kosba et al. [26] also design a modular notational system that is intuitive and factors out tedious but common features inside functionality and protocol wrappers (e.g., modeling of time, pseudonyms, adversarial reordering of messages, a global ledger). The advantages of adopting Kosba et al.'s notational system are these: the blockchain contracts and user-side protocols are intuitive on their own and they are endowed with precise, formal meaning when we apply the blockchain wrappers.

Technical subtleties. While Kosba et al.'s formal blockchain model is applicable for the most part, we point out a subtle mismatch between their formal blockchain model in [26] and the real-world instantiation of blockchains such as Ethereum (and Bitcoin for that matter). The design of Town Crier is secure in a slightly modified version of the blockchain model that more accurately reflects the real-world Ethereum instantiation of a blockchain.

Recall that one tricky detail for the gas-aware version of the Town Crier contract arises from the need to deal with with **Deliver** arriving after **Cancel**. In the formal blockchain model proposed by Kosba et al. [26], we can easily get away with this issue by introducing a timeout parameter $\mathsf{T}_{\text{timeout}}$ that the requester attaches to each datagram request. If the datagram fails to arrive before $\mathsf{T}_{\text{timeout}}$, the requester can call **Cancel** any time after $\mathsf{T}_{\text{timeout}} + \Delta T$. On the surface, this seems to ensure that no **Deliver** will be invoked after **Cancel** assuming Town Crier is honest.

However, we did not adopt this approach due to a technical subtlety that arises in this context—again, the fact that the Ethereum blockchain does not perfectly match the formal blockchain model specified by Kosba et al [26]. Specifically, the blockchain model by Kosba et al. assumes that every message (i.e., transaction) will be delivered to the blockchain by the end of each epoch and that the adversary cannot drop any message. In practice, however, Ethereum adopts a dictatorship strategy in the mining protocol, and the winning miner for an epoch can censor transactions for this specific epoch, and thus effectively this transaction will be deferred to later epochs. Further, in case there are more incoming transactions than the block size capacity of Ethereum, a backlog of transactions will build up, and similarly in this case there is also guaranteed ordering of backlogged transactions. Due to these considerations, we defensively design our Town Crier contract such that gas neutrality is attained for Town Crier even if the **Deliver** transaction arrives after **Cancel**.

C Proofs of Security

This section contains the proofs of the theorems we stated in Section 6.

Theorem 1 (Authenticity)**.** *Assume that Σ_{sgx} and Σ are secure signature schemes. Then, the full TC protocol achieves authenticity of data feed under Definition 1.*

Proof. 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-negligible 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 pk_{TC} as the SGX functionality $\mathcal{F}_{\text{sgx}}$. 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 $\mathcal{F}_{\text{sgx}}$ 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.

□

Lemma 1. *$\mathcal{C}_{TC}$ will never attempt to send money in* **Deliver** *or* **Cancel** *that were not deposited with the given* id.

Proof. First we note that there are only three lines on which $\mathcal{C}_{TC}$ sends money: (2), (3), and (5). Second, for a request id, $\$\mathtt{f}$ is deposited. Third, because isCanceled[id] is only set immediately prior to line (5) and line (2) is only reachable if isCanceled[id] is set, it is impossible to reach line (2) without reaching line (5).

We now consider cases based on which of lines (3) and (5) are reached first (since at least one must be reached to send any money).

- *Line* (5) *is reached first.* In this case, line (5) sends $\$\mathtt{f} - \$\mathtt{G}_{\emptyset}$ and allows $\$\mathtt{G}_{\emptyset}$ to remain. Future calls to **Cancel** with this id will fail the isCanceled[id] not check assertion, so line (5) can never be reached again with this id. If $\mathcal{W}_{TC}$ invokes **Deliver** after this point, the first such invocation will satisfy the predicate on line (1) and proceed to set isDelivered[id] and reach line (2). Any future entries to **Deliver** with id will fail to satisfy the predicate on line (1) and then fail an assertion and abort prior to line (3). Since line (2) sends $\$\mathtt{G}_{\emptyset}$, the total money sent in connection with id is $(\$\mathtt{f} - \$\mathtt{G}_{\emptyset}) + \$\mathtt{G}_{\emptyset} = \$\mathtt{f}$.
- *Line* (3) *is reached first.* In this case, line (3) send the full $\$\mathtt{f}$ immediately after setting isDelivered[id]. With isDelivered[id] set, any call to **Cancel** with id will fail an assertion prior to line (5) and any future call to **Deliver** with id will fail to satisfy the predicate on line (1) and also fail an assertion prior to reaching line (3). Thus no further money will be distributed in connection with id.

□

Theorem 2 (Gas neutrality for Town Crier)**.** *If the TC Relay is honest, Town Crier's wallet $\mathcal{W}_{TC}$ will have at least $\$\mathtt{G}_{max}$ remaining after each* **Deliver** *call.*

Proof. By assumption, $\mathcal{W}_{TC}$ is seeded with at least $\$\mathtt{G}_{\text{max}}$ money. Thus it suffices to prove that, given an honest Relay, $\mathcal{W}_{TC}$ will have at least as much money after invoking **Deliver** as it did before.

An honest Relay will never ask for a response for the same id more than once. **Deliver** only responds to messages from $\mathcal{W}_{TC}$, and isDelivered[id] is only set inside **Deliver**, so therefore we know that isDelivered[id] is not set for this id. We now consider the case where isCanceled[id] is set upon invocation of **Deliver** and the case where it is not.

- isCanceled[id] *not set:* In this case the predicate on line (1) of the protocol returns `false`. Because the Relay is honest, id exists and $\mathsf{params} = \mathsf{params}'$. The enclave always provides $\$\mathtt{g}_{\text{dvr}} = \$\mathtt{G}_{\text{max}}$ (which it has by assumption) and **Request** ensures that $\$\mathtt{f} \leq \$\mathtt{G}_{\text{max}}$. Thus, coupled with the knowledge that isDeliver[id] is not set, all assertions pass and we progress through lines (3) and (4). Now we must show that at line (3) $\mathcal{C}_{TC}$ had $\$\mathtt{f}$ to send and that the total gas spend to execute **Deliver** does not exceed $\$\mathtt{f}$.

  To see that $\mathcal{C}_{TC}$ had sufficient funds, we note that upon entry to **Deliver**, both isDelivered[id] and isCanceled[id] must have been unset. The first we showed above. The second is because, given the first, if isCanceled[id] were set, the predicate on line (1) would have returned true, sending execution on a path that would not encounter (4). This means that line (5) was never reached because the preceding line sets isCanceled[id]. Because (2), (3), and (5) are the only lines that remove money from $\mathcal{C}_{TC}$ and $\$\mathtt{f}$ was deposited as part of **Request**, it must be the case that $\$\mathtt{f}$ is still in the contract.

  To see how much gas is spent, we first note that $\$\mathtt{G}_{\text{min}}$ is defined to be the amount of gas needed to execute **Deliver** along this execution path not including line (4). Since $\$\mathtt{g}_{\text{clbk}}$ is set to $\$\mathtt{f} - \$\mathtt{G}_{\text{min}}$ and line (4) is limited to using $\$\mathtt{g}_{\text{clbk}}$ gas, the total gas spent on this execution of **Deliver** is at most $\$\mathtt{G}_{\text{min}} + (\$\mathtt{f} - \$\mathtt{G}_{\text{min}}) = \$\mathtt{f}$.
- isCanceled[id] *is set:* Here the predicate on line (1) returns `true`. Along this execution path $\mathcal{C}_{TC}$ sends $\mathcal{W}_{TC}$ $\$\mathtt{G}_{\emptyset}$ and quickly returns. $\$\mathtt{G}_{\emptyset}$ is defined as the amount of gas necessary to execute this execution path, so we need only show that $\mathcal{C}_{TC}$ has $\$\mathtt{G}_{\emptyset}$ available to send.

  Because isCanceled[id] is set, it must be the case that Cancel was invoked with id and reached line (5). Gas exhaustion in Cancel is not a concern because it would abort and revert the entire invocation. This is only possible if the data retrieval and all assertions in Cancel succeed. In particular, this means that id corresponds to a valid request which deposited $\$\mathtt{f}$. Line (5) returns $\$\mathtt{f} - \$\mathtt{G}_{\emptyset}$ to $\mathcal{C}_U$, but it leaves $\$\mathtt{G}_{\text{min}}$ from the original $\$\mathtt{f}$. Moreover, if **Cancel** is invoked multiple times with the same id, all but the first will fail due to the assert that isCanceled[id] is not set and the fact that any invocation that reaches (5) will set isCanceled for that id. Since

only lines (2), (3), and (5) can remove money from $\mathcal{C}_{TC}$ and line (3) will never be called in this case, there will still be exactly $\$\mathtt{G}_{\text{min}}$ available when this invocation of **Deliver** reaches line (2).

□

Theorem 3 (Fair Expenditure for Honest Requester)**.** *For any* params *and* callback*, let* $\$\mathtt{G}_{req}$ *and* $\$\mathtt{F}$ *be the respective values chosen by an honest requester for* $\$\mathtt{g}_{req}$ *and* $\$\mathtt{f}$ *when submitting the request* $(\mathsf{params}, \mathsf{callback}, \$\mathtt{f}, \$\mathtt{g}_{req})$*. For any such request submitted by an honest user* $\mathcal{C}_U$*, one of the following holds:*

- callback *is invoked with a valid datagram matching the request parameters* params *and the requester spends at most* $\$\mathtt{G}_{req} + \$\mathtt{G}_{cncl} + \$\mathtt{F}$.
- *The requester spends at most* $\$\mathtt{G}_{req} + \$\mathtt{G}_{cncl} + \$\mathtt{G}_{\emptyset}$.

Proof. $\mathcal{C}_U$ is honest, so she will spend $\$\mathtt{G}_{\text{req}}$ to invoke **Request**$(\mathsf{params}, \mathsf{callback}, \$\mathtt{F})$. Ethereum does not allow money to change hands without the payer explicitly sending money. Therefore we must only examine the explicit function invocations and monetary transfers initiated by $\mathcal{C}_U$ in connection with the request. It is impossible for $\mathcal{C}_U$ to lose more money than she gives up in these transactions even if TC is malicious.

- *Request Delivered:* If protocol line (4) is reached, then we are guaranteed that $\mathsf{params} = \mathsf{params}'$ and $\$\mathtt{g}_{\text{dvr}} \geq \$\mathtt{F}$. By Theorem 1, the datagram must therefore be authentic for params. Because $\$\mathtt{F}$ is chosen honestly for callback, $\$\mathtt{F} - \$\mathtt{G}_{\text{min}}$ is enough gas to execute callback, so callback will be invoked with a datagram that is a valid and matches the request parameters.

  In this case, the honest requester will have spent $\$\mathtt{G}_{\text{req}}$ to invoke **Request** and $\$\mathtt{F}$ in paying TC's cost for **Deliver**. The requester may have also invoked **Cancel** at most once at the cost of $\$\mathtt{G}_{\text{cncl}}$. While $\mathcal{C}_U$ may not receive any refund due to **Cancel** aborting, $\mathcal{C}_U$ will still have spent at most $\$\mathtt{G}_{\text{req}} + \$\mathtt{G}_{\text{cncl}} + \$\mathtt{F}$.

- *Request not Delivered:* The request not being delivered means that line (4) is never reached. This can only happen if **Deliver** is never called with a valid response or if isCanceled[id] is set before deliver is called. The only way to set isCanceled[id] is for $\mathcal{C}_U$ to invoke **Cancel** with isDelivered[id] not set. If deliver is not executed, we assume that an honest requester will eventually invoke **Cancel**, so this case will always reach line (5). When line (5) is reached, then $\mathcal{C}_U$ will have spent $\$\mathtt{G}_{\text{req}} + \$\mathtt{F}$ while executing **Request**, and spent $\$\mathtt{G}_{\text{cncl}}$ in **Cancel** and will attempt to retrieve $\$\mathtt{F} - \$\mathtt{G}_{\emptyset}$.

  The retrieval will succeed because $\mathcal{C}_{TC}$ will always have the funds to send $\mathcal{C}_U$ $\$\mathtt{F} - \$\mathtt{G}_{\emptyset}$. To see this, Lemma 1 allows us to consider only **Deliver** and **Cancel** calls associated with id.

  Since line (5) is reached, it must be the case the isDelivered[id] is not set. This means that neither lines (2) nor (3) were reached since the line before each sets isDelivered[id]. The lines preceding those two and (5) are the only lines that remove money from the contract. Line (5) cannot have been reached before because $\mathcal{C}_U$ is assumed to be honest, so she will not invoke **Cancel** twice for the same request and if any other user invokes **Cancel** for this request, the $\mathcal{C}_U = \mathcal{C}'_U$ assertion will fail and the invocation will abort before line (5). Because none of (2), (3), or (5) has been reached before and $\mathcal{C}_U$ deposited $\$\mathtt{F} > \$\mathtt{G}_{\text{min}} > \$\mathtt{G}_{\emptyset}$ on **Request**, it must be the case that $\mathcal{C}_{TC}$ has $\$\mathtt{F} - \$\mathtt{G}_{\emptyset}$ left.

  This means the total expenditure in this case will be

$$\begin{aligned}&\$\mathtt{G}_{\text{req}} + \$\mathtt{G}_{\text{cncl}} + \$\mathtt{F} - (\$\mathtt{F} - \$\mathtt{G}_{\emptyset})\\ &\qquad = \$\mathtt{G}_{\text{req}} + \$\mathtt{G}_{\text{cncl}} + \$\mathtt{G}_{\emptyset}.\end{aligned}$$

□

D Applications and Code Samples

We now elaborate on the applications described in Section 7 and we show a short Solidity code sample for one of these applications, to demonstrate first-hand what a requester contract would look like to call Town Crier's authenticated data feed service.

Financial derivative (CashSettledPut)**.** Financial derivatives are among the most commonly cited smart contract applications, and exemplify the need for a data feed on financial instruments. We implemented an example contract CashSettledPut for a *cash-settled put option*. This is an agreement for one party to buy an asset from the other at an agreed upon price on or before a particular date. It is "cash-settled" in that the sale is implicit, i.e. no asset changes hands, only cash reflecting the asset's value. In our implementation, the issuer of the option specifies a strike price P_S, expiration date, unit price P_U, and maximum number of units M she is willing to sell. A customer may send a request to the contract specifying the number X of option units to be purchased and containing the associated fee $(X \cdot P_U)$. A customer may then exercise the option by sending another request prior to the expiration date. CashSettledPut calls TC to retrieve the closing price P_C of the underlying instrument on the day the option was exercised, and pays the customer $X \cdot (P_S - P_C)$. To ensure sufficient funding to pay out, the contract must be endowed with ether value at least $M \cdot P_S$.

In Figure 13 we describe the protocol for CashSettledPut. We omit the full source code due to length and complexity.

CashSettledPut **blockchain contract**

Constants

T_{stock} := Town Crier stock info request type
$\$F_{TC}$:= fee payed to TC for datagram delivery

Functions

Init: On recv $(\mathcal{C}_{TC}, \text{ticker}, P_S, P_U, M, \text{expr}, \$f)$ from $\mathcal{W}_{\text{issuer}}$
Assert $\$f = (P_S - P_U) \cdot M + \$F_{TC}$
Save all inputs and $\mathcal{W}_{\text{issuer}}$ to storage.

Buy: On recv $(X, \$f)$ from $\mathcal{W}_U$:
Assert isRecovered not set
and timestamp $<$ expr
and $\mathcal{W}_{\text{buyer}}$ not set
and $X \leq M$
and $\$f = (X \cdot P_U)$
Set $\mathcal{W}_{\text{buyer}} = \mathcal{W}_U$
Save X to storage
Send $(P_S - P_U)(M - X)$ to $\mathcal{W}_{\text{issuer}}$
// Hold $P_S \cdot X + \$F_{TC}$

Put: On recv () from $\mathcal{W}_{\text{buyer}}$:
and timestamp $<$ expr
and isPut not set
Set isPut
params := $[T_{\text{stock}}, \text{ticker}]$
callback := `this`.**Settle**
$\mathcal{C}_{TC}$.**Request**(params, callback, $\$F_{TC}$)

Settle: On recv (id, P) from $\mathcal{C}_{TC}$:
If $P \geq P_S$
Send $P_S \cdot X$ to $\mathcal{W}_{\text{issuer}}$
Return
Send $(P_S - P)X$ to $\mathcal{W}_{\text{buyer}}$
Send all money in contract to $\mathcal{W}_{\text{issuer}}$
Send $P \cdot X$ to $\mathcal{W}_{\text{issuer}}$

Recover: On recv () from $\mathcal{W}_{\text{issuer}}$:
and isPut not set
and isRecovered not set
and ($\mathcal{W}_{\text{buyer}}$ not set
or timestamp $\geq$ expr)
Set isRecovered
Send all money in contract to $\mathcal{W}_{\text{issuer}}$

Figure 13: The CashSettledPut application contract

Flight insurance (FlightIns)**.** Flight insurance indemnifies a purchaser should her flight be delayed or canceled. We have implemented a simple flight insurance contract called FlightIns. Our implementation showcases TC's *private-datagram* feature to address an obvious concern: customers may not wish to reveal their travel plans publicly on the blockchain.

An insurer stands up FlightIns with a specified policy fee, payout, and lead time ΔT. (ΔT is set large enough to ensure that a customer can't anticipate flight cancellation or delay due to weather, etc.) To purchase a policy, a customer sends the FlightIns a ciphertext C under the TC's pubic key pk_{TC} of the ICAO flight number FN and scheduled time of departure T_D for her flight, along with the policy fee. FlightIns sends TC a private-datagram request containing the current time T and the ciphertext C. TC decrypts C and checks that the lead time meets the policy requirement, i.e., that $T_D - T \geq \Delta T$. TC then scrapes a flight information data source several hours after T_D to check the flight status, and returns to FlightIns predicates on whether the lead time was valid and whether the flight has been delayed or canceled. If both predicates are true, then FlightIns returns the payout to the customer. Note that FN is never exposed in the clear.

Despite the use of private datagrams, FlightIns as described here still poses a privacy risk, as the *timing* of the predicate delivery by TC leaks information about T_D, which may be sensitive information; this, and the fact that the payout is publicly visible, could also indirectly reveal FN. FlightIns addresses this issue by including in the private datagram request another parameter $t > T_D$ specifying the time at which predicates should be returned. By randomizing t and making $t - T_D$ sufficiently large, FlightIns can substantially reduce the leakage of timing information.

In Figure 14 we include a full implementation of FlightIns in Solidity.

Steam Marketplace (SteamTrade)**.** Steam [5] is an online gaming platform that supports thousands of games and maintains its own marketplace, where users can trade, buy, and sell games and other virtual items. We implement a contract for the sale of games and items for ether that showcases TC's support for custom datagrams through the use of Steam's access-controlled API.

A user intending to sell items creates a contract SteamTrade with her Steam account number ID_S, a list L of items for sale, a price P, and a ciphertext C under the TC's public key pk_{TC} of her Steam API key. In order to purchase the list of items, a buyer first uses a Steam client to create a trade offer requesting each item in L. The buyer then submits to SteamTrade her Steam account number ID_U, a length of time T_U indicating how long the seller has to respond to the offer, and an amount of ether equivalent to the price P. SteamTrade sends TC a custom datagram containing the current time T, ID_U, T_U, L, and the encrypted API key C. TC decrypts C to obtain the API key, delays for time T_U, then retrieves all trades between the two accounts using the provided API key within that time period. TC verifies whether or not a trade exactly matching the items in L successfully occurred between the two accounts and returns the result

```
// A simple flight insurance contract using Town Crier's private datagram.
contract FlightIns {
  uint   constant TC_REQ_TYPE = 0;
  uint   constant TC_FEE      = (35000 + 20000) * 5 * 10**10;
  uint   constant FEE         = 10**18;     //   $5 in wei
  uint   constant PAYOUT      = 2 * 10**19; // $200 in wei
  uint32 constant MIN_DELAY   = 30;

  // The function identifier in Solidity is the first 4 bytes
  // of the sha3 hash of the functions' canonical signature.
  // This contract's callback is bytes4(sha3("pay(uint64,bytes32)"))
  bytes4 constant CALLBACK_FID = 0x3d622256;

  TownCrier tc;
  address[2**64] requesters;

  // Constructor which sets the address of the Town Crier contract.
  function FlightIns(TownCrier _tc) public {
    tc = _tc;
  }

  // A user can purchase insurance through this entry point.
  // encFN is an encryption of the flight number and date
  // as well as the time when Town Crier should respond to the request.
  function insure(bytes32[] encFN) public {
    if (msg.value != FEE) return;

    // Adding money to a function call involves calling ".value()"
    // on the function itself before calling it with arguments.
    uint64 requestId =
      tc.request.value(TC_FEE)(TC_REQ_TYPE, this, CALLBACK_FID, encFN);
    requesters[requestId] = msg.sender;
  }

  // This is the entry point for Town Crier to respond to a request.
  function pay(uint64 requestId, bytes32 delay) public {
    // Check that this is a response from Town Crier
    // and that the ID is valid and unfulfilled.
    address requester = requesters[requestId];
    if (msg.sender != address(tc) || requester == 0) return;

    if (uint(delay) >= MIN_DELAY) {
      address(requester).send(PAYOUT);
    }
    requesters[requestId] = 0;
  }
}
```

Figure 14: Solidity code for the FlightIns application contract.

to SteamTrade. If such a trade occurred, SteamTrade sends the buyer's ether to the seller's account. Otherwise the buyer's ether is refunded.

In Figure 15 we describe the protocol for CashSettledPut. We again omit the full source code due to length and complexity.

SteamTrade **blockchain contract**

Constants

T_{steam} := Town Crier Steam trade request type
$\$F_{TC}$:= fee payed to TC for datagram delivery

Functions

Init: On recv $(\mathcal{C}_{TC}, ID_S, encAPI_S, List_I, P)$ from $\mathcal{W}_{\text{seller}}$:
Save all inputs and $\mathcal{W}_{\text{seller}}$ to storage.

Buy: On recv $(ID_U, T_U, \$f)$ from $\mathcal{W}_U$:
Assert $\$f = P$
params := $[encAPI_S, ID_U, T_U, LIST_I]$
callback := `this`.**Pay**
id := $\mathcal{C}_{TC}$.**Request**(params, callback, $\$F_{TC}$)
Store $(\text{id}, \mathcal{W}_U)$

Pay: On recv $(\text{id}, status)$ from $\mathcal{C}_{TC}$:
Retrieve and remove stored $(\text{id}, \mathcal{W}_U)$
// Abort if not found
If $status > 0$
Send $\$F_{\text{price}}$ to $\mathcal{W}_{\text{seller}}$
Else
send $\$F_{\text{price}}$ to $\mathcal{W}_U$

Figure 15: The FlightIns application contract

E Future Work

We plan to develop TC after its initial deployment and expect it to evolve to incorporate a number of additional features. These fall into two categories: (1) Expanding the security model to address threats outside the scope of the initial version and (2) Extending the functionality of TC. Here we briefly discuss a few of these extensions.

E.1 Expanding TC threat model

- *Freeloading protection.* Serious concerns have arisen in the Ethereum community about "parasite contracts" that forward or resell datagrams—particularly those from fee-based data feeds [34]. We plan to deploy a novel mechanism in TC to address this concern. Suppose contract $\mathcal{C}_U$ involves a set of parties / users $U = \{U_i\}_{i=1}^n$. Each player U_i generates an individual share $(\mathsf{sk}_i, \mathsf{pk}_i)$ of a global keypair $(\mathsf{pk}, \mathsf{sk})$, where $\mathsf{sk} = \sum_{i=1} \mathsf{sk}_i$ and $\mathsf{pk} = \prod_{i=1} \mathsf{pk}_i$, and communicates a ciphertext $E_{\mathsf{pk}_{TC}}[\mathsf{sk}_i]$ to $\mathcal{C}_{TC}$, e.g., by including it in a datagram request. Players then jointly set up under public key pk a wallet $\mathcal{W}^*$ for datagram transmission by $\mathcal{C}_{TC}$.

  Thanks to the homomorphic properties of ECDSA, $\mathcal{C}_{TC}$ can compute sk (non-interactively) and send datagrams from $\mathcal{W}^*$. But the users U collaboratively *can also compute* sk *and send messages from* $\mathcal{W}^*$. Consequently, while each user U_i can individually be assured that a datagram sent to $\mathcal{C}_U$ by $\mathcal{C}_{TC}$ from $\mathcal{W}^*$ is valid (as U_i didn't collude in its creation), other players cannot determine whether a datagram was produced by $\mathcal{C}_{TC}$ or U, and thus whether or not it is valid. Such a *source-equivocal datagram* renders data from parasite contracts less trustworthy and thus less attractive.

- *Traffic-analysis protection.* As noted in the body of the paper, the Relay can observe the pattern of data source accesses made by TC. By correlating with activity in $\mathcal{C}_{TC}$, an adversarial Relay can thus infer the data source targeted by private datagrams, as well as the timing. This adversary can potentially also perform traffic analysis on the Enclave's HTTPS requests. (See, e.g., [19].) In the example contract FlightIns in Section 7, this issue is partially addressed through the insertion of random delays into TC responses. We intend to develop a comprehensive approach to mitigating traffic analysis in TC. This approach will include, for the problem of traffic analysis of web scraping, the incorporation of facilities in the Enclave to make chaff or decoy data requests, i.e., false requests, to both the true data source and well as non-target data sources.

- *Revocation support.* Revocation of two forms can impact the TC service.

  First, the certificates of data sources may be revoked. To address this issue, given its ability to establish external HTTPS connections, TC could easily make use of Online Certificate Status Protocol (OCSP) certificate checking. This functionality would amount to an additional form of web scraping, and could be executed in parallel with web scraping to support datagram requests, resulting in minimal additional latency.

  Second, an SGX host could become compromised, prompting revocation of its EPID signatures by Intel. The Intel Attestation Service (IAS) will reportedly provide support for online attestation verification and thus for revocation. Conveniently, clients use the IAS when checking the attestation σ_{att}, so no modification to TC is required to support the service.

- *SLAs.* Were TC to be deployed as a fee-for-service system, requesters might wish to see Service Level Agreements (SLAs) enforced. A temporary outage on a TC host or a malicious Relay could cause a delay in datagram delivery, potentially with aftereffects in relying

contracts. An SLA could be implemented as an indemnity paid to a contract if a datagram is not delivered within a specified period of time. This feature could itself be implemented, for example, as an entry point in $\mathcal{C}_{TC}$. (Naturally care would be required to ensure that $\mathcal{C}_{TC}$ holds funds sufficient for payout in the case of a general delivery failure.)

- *Hedging against SGX compromise.* We discussed in Section 5.3 and demonstrated in Section 8.4 how TC can support majority voting across SGX hosts and/or data sources to hedge against failures in either (outside TC's basic security model) via majority voting. It would be possible to reduce the latency and gas costs of such voting with design enhancements to TC. Specifically, for the case of SGX voting, we plan to investigate a scheme in which consensus on a datagram value X is reached off-chain among SGX-enable TC hosts via Byzantine consensus. These hosts may then use a threshold digital signature scheme to sign the datagram response from $\mathcal{W}_{TC}$. To ensure that the response is delivered to the blockchain, each participating host can monitor the blockchain and itself transmit the response in the case of an observed delivery failure. This approach will largely eliminate the incremental gas cost of majority voting across SGX instances.

E.2 Expanding TC functionality

- *New opcodes.* Ethereum's developers [6] have indicated an intention to expand the range of supported cryptographic primitives in Ethereum and stated that they are amenable to the authors' suggestion of incorporating opcodes supporting Intel's EPID in particular, which would enable efficient attestation verification within the blockchain.

- *Migration to data-source feeds.* Ultimately, we envision that data sources may wish themselves to serve as authenticated data feeds. To do so, they could simply stand up TC as a front end. As a first step along this path, however, an independent TC service might provide support for XML-labelled data from data sources, enabling more accurate and direct scraping and intentional identification of what data should be served. We plan to build support for such explicit data labeling into TC should this approach prove attractive to data sources.

- *Generalized custom datagrams.* In our example smart contract SteamTrade, we demonstrated a custom datagram that is essentially hardwired: It employs a user's credentials to scrape her individual online account. A more general approach would be to allow contract owners to specify their own generalized functionalities–scrapers and/or confidential contract modules—as general purpose code, achieving a data-source-enriched emulation of private contracts as in Hawk [26], but with much lower resource requirements. Furnishing large custom datagrams on the blockchain would be prohibitively expensive, but off-chain loading of code would be quite feasible. Of course, many security and confidentiality considerations arise in a system that allows users to deploy arbitrary code, giving rise to programming language challenges that deployment of this feature in TC would need to address.