Town Crier: An Authenticated Data Feed for Smart Contracts

Fan Zhang[1,3] fanz@cs.cornell.edu
Ethan Cecchetti[1,3] ethan@cs.cornell.edu
Kyle Croman[1,3] kcroman@cs.cornell.edu
Ari Juels[2,3] juels@cornell.edu
Elaine Shi[1,3] rs2358@cornell.edu

[1]Cornell University [2]Cornell Tech, Jacobs Institute [3]Initiative for CryptoCurrencies and Contracts

ABSTRACT

Smart contracts are programs that execute autonomously on blockchains. Their key envisioned uses (e.g. financial instruments) require them to consume data from outside the blockchain (e.g. stock quotes). Trustworthy *data feeds* that support a broad range of data requests will thus be critical to smart contract ecosystems.

We present an authenticated data feed system called *Town Crier* (TC). TC acts as a bridge between smart contracts and existing web sites, which are already commonly trusted for non-blockchain applications. It combines a blockchain front end with a trusted hardware back end to scrape HTTPS-enabled websites and serve source-authenticated data to relying smart contracts. TC also supports confidentiality; it enables private data requests with encrypted parameters and secure use of user credentials to scrape access-controlled online data sources.

We describe TC's design principles and architecture and report on an implementation that uses Intel's recently introduced Software Guard Extensions (SGX) to furnish data to the Ethereum smart contract system. We formally model TC and define and prove its basic security properties in the Universal Composibility (UC) framework. Our results include definitions and techniques of general interest relating to resource consumption (Ethereum's "gas" fee system) and TCB minimization. We also report on experiments with three example applications.

We plan to launch TC soon 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 render legal agreements more precise, pervasive, and efficiently executable. Szabo, who popularized the term "smart contact" in a seminal 1994 essay [36], 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 [30] provide key technical underpinnings for smart contracts: direct control of money by programs and fair, automated code execution through the decentralized consensus mechanisms underlying blockchains. The recently launched Ethereum [14, 38] supports Turing-complete code and thus fully expressive self-enforcing decentralized 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—such as financial instruments—additionally require access to *data about real-world state and events*.

Data feeds (also known as "oracles") aim to meet this need. Very simply, data feeds are contracts on the blockchain that serve data requests by other contracts [14, 38]. A few data feeds exist for Ethereum today that source data from trustworthy websites, but provide no assurance of correctly relaying such data beyond the reputation of their operators (typically individuals or small entities). HTTPS connection to a trustworthy website would seem to offer a solution, but smart contracts lack network access, and HTTPS does not digitally sign data for out-of-band verification. The lack of a substantive ecosystem of trustworthy data feeds is frequently cited as critical obstacle to the evolution of Ethereum and decentralized smart contracts in general [21].

Town Crier. We introduce a system called *Town Crier* (TC) that addresses this challenge by providing 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 relying contracts on the blockchain as concise pieces of data (e.g. stock quotes) called *datagrams*. TC uses a novel combination of Software Guard Extensions (SGX), Intel's recently released trusted hardware capability, and a smart-contract front end. 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.

The smart-contract front end of 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 verify the correctness of X in such a datagram assuming trust only in the security of SGX, the (published) TC code, and the validity of source data in the specified interval of time.

Another critical barrier to smart contract adoption is the lack of *confidentiality* in today's ecosystems; all blockchain state is publicly visible, and existing data feeds publicly expose requests. TC provides confidentiality by supporting *private* datagram requests, in which the parameters are en-

crypted under a TC public key for ingestion in TC's SGX enclave and are therefore concealed on the blockchain. TC also supports *custom* datagram requests, which securely access the online resources of requesters (e.g. online accounts) by ingesting encrypted user credentials, permitting TC to securely retrieve access-controlled data.

We designed and implemented TC as a complete, highly scalable, end-to-end system that offers formal security guarantees at the cryptographic protocol level. TC runs on real, SGX-enabled host, as opposed to an emulator (e.g. [10,33]). We plan to launch a version of TC as an open-source, production service atop Ethereum, pending the near-future availability of the Intel Attestation Service (IAS), which is needed to verify SGX attestations.

Technical challenges. Smart contracts execute in an adversarial environment where parties can reap financial gains by subverting the contracts or services on which they rely. Formal security is thus vitally important. We adopt a rigorous approach to the design of Town Crier by modeling it in the Universal Composibility (UC) framework, building on [28, 35] to achieve an interesting formal model that spans a blockchain and trusted hardware. We formally define and prove that TC achieves the basic property of datagram *authenticity*—informally that TC faithfully relays current data from a target website. We additionally prove *fair expenditure* for an honest requester, informally that the fee paid by a user contract calling TC is at most a small amount to cover the operating costs of the TC service, even if the TC host is malicious.

Another contribution of our work is introducing and showing how to achieve two key security properties: *gas sustainability* and *trusted computing base (TCB) code minimization* within a new TCB model created by TC's combination of a blockchain with SGX.

Because of the high resource costs of decentralized code execution and risk of application-layer denial-of-service (DoS) attacks, Ethereum includes an accounting resource called *gas* to pay for execution costs. Informally, *gas sustainability* means that an Ethereum service never runs of out of gas, a general and fundamental availability property. We give a formal definition of gas sustainability applicable to any Ethereum service, and prove that TC satisfies it.

We believe that the combination of blockchains with SGX introduced in our work will prove to be a powerful and general way to achieve confidentiality in smart contract systems and network them with off-chain systems. This new security paradigm, however, introduces a hybridized TCB that spans components with different trust models. We introduce techniques for using such a hybridized TCB securely while *minimizing the TCB code size*. In TC, we show how to avoid constructing an authenticated channel from the blockchain to the enclave—bloating the enclave with an Ethereum client—by instead authenticating enclave outputs on the blockchain. We also show how to minimize on-chain signature-verification code. These techniques are general; they apply to any use of a similar hybridized TCB.

Other interesting smaller challenges arise in the design of TC. One is deployment of TLS in an enclave. Enclaves lack networking capabilities, so TLS code must be carefully partitioned between the enclave and untrusted host environment. Another is hedging in TC against the risk of compromise of a website or single SGX instance, which we accomplish with various modes of majority voting: among multiple websites offering the same piece of data (e.g. stock price) or among multiple SGX platforms.

Applications and performance. 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 explore three examples that demonstrate TC's capabilities: (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 user accounts.

Our experiments with these three applications show that TC is highly scalable. Running on just a single SGX host, TC achieves throughputs of 15-65 tx/sec. TC is easily parallelized across many hosts, as separate TC hosts can serve requests with no interdependency. (For comparison, Ethereum handles less than 1 tx/sec today and recent work [19] suggests that Bitcoin can scale safely to no more 26 tx/sec with reparametrization.) For these same applications, experimental response times for datagram requests range from 192-1309 ms—much less than an Ethereum block interval (12 seconds on average). These results suggest that a few SGX-enabled hosts can support TC data feed rates well beyond the global transaction rate of a modern decentralized blockchain.

Contributions. We offer the following contributions:

- We introduce and report on an end-to-end implementation of Town Crier, an authenticated data feed system that addresses critical barriers to the adoption of decentralized smart contracts. TC combines a smart-contract front end in Ethereum and an SGX-based trusted hardware back end to: (1) Serve authenticated data to smart contracts without a trusted service operator and (2) Support *private* and *custom* data requests, enabling encrypted requests and secure use of access-controlled, off-chain data sources. We plan to launch a version of TC soon as an open-source service.
- We formally analyze the security of TC within the Universal Composibility (UC) framework, defining functionalities to represent both on-chain and off-chain components. We formally define and prove the basic properties of datagram *authenticity* and *fair expenditure* as well as *gas sustainability*, a fundamental availability property for any Ethereum service.
- We introduce a hybridized TCB spanning the blockchain and an SGX enclave, a powerful new paradigm of trustworthy system composition. We present generic techniques that help shrink the TCB code size within this model as well as techniques to hedge against individual SGX platform compromises.
- We explore three TC applications that show TC's ability to support a rich range of services well beyond those in Ethereum today. Experiments with these applications also show that TC can easily meet the latency and throughput requirements of modern decentralized blockchains.

Due to space constraints, a number of details on formalism, proofs, implementation, and applications are relegated to the paper appendices with pointers in the paper body. Appendices may be found in the supplementary materials.

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) [8, 22, 29, 31] 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.

An 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 [12]. This choice of primitive is significant in our design of Town Crier, as EPID is a proprietary signature scheme not supported natively 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 [14, 38].

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 under $15 U.S. [1].) To prevent DoS attacks, prevent inadvertent infinite looping within contracts, and generally control network resource expenditure, 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 and gas is still spent.

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).

As we discuss in Section 5.1, the management of gas is critical to the availability of TC (and other Ethereum-based services) in the face of malicious users.

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. ARCHITECTURE AND SECURITY MODEL

Town Crier includes three main components: The TC Contract ($\mathcal{C}_{TC}$), the Enclave (whose code is denoted by $\mathsf{prog}_{\text{encl}}$), 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

[1]This model is a simplification: SGX is known to expose some internal enclave state to the OS [18]. 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.

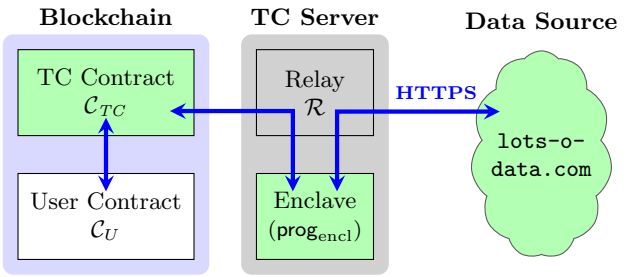

Figure 1: Basic Town Crier architecture. Trusted components are depicted in green.

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 $\mathsf{prog}_{\mathrm{encl}}$. 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 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 $\mathcal{C}_{TC}$. 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 Enclave attestations. 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 between data sources (HTTPS-enabled websites) and 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 will 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., $\mathsf{prog}_{\mathrm{encl}}$, whose source code is published for clients, correctly executes the 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 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 Enclave's behavior.) Thus the Relay is subsumed by an adversary that controls the network.

4. TC PROTOCOL OVERVIEW

We now outline the protocol of TC at a high level. The basic structure 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 the Enclave 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 overview of the protocol structure. Then we enumerate the data flows in TC. Finally, we present the framework for modeling SGX as ideal functionalities inspired by the universal-composability (UC) framework.

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 callback in m_1 indicates the entry point to which the datagram is to be returned. While callback need not be in $\mathcal{C}_U$, we assume it is for simplicity.

$\mathcal{C}_{TC}$ generates a fresh unique id and forwards $m_2 = (\mathsf{id}, \mathsf{params})$ to the Enclave. In response it receives $m_3 = (\mathsf{id}, \mathsf{params}, \mathsf{data})$ 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 request and response and, if they match, forwards data to the callback entry point 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.

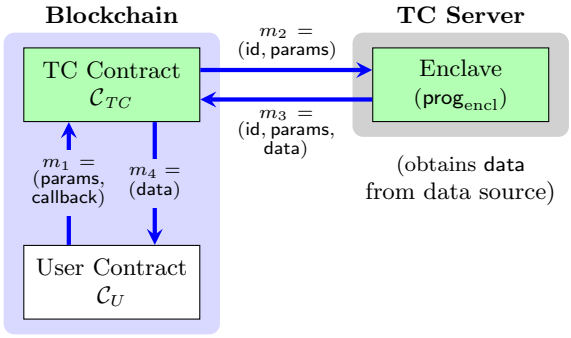

Figure 2: Data flows in datagram processing.

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 later how TC uses a layer of indirection to 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 $\mathsf{prog}_{\mathrm{encl}}$. To achieve this goal, we first present a model that abstracts away the details of SGX, helping to simplify our protocol presentation and security proofs. We also explain how we use the clock in SGX. Our discussion draws on formalism for SGX from Shi et al. [35].

Formal model and notation. We adopt a formal abstraction of Intel SGX proposed by Shi et al. [35]. Following the UC and GUC paradigms [15–17], 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 of the TC Relay). As described in Fig. 3, upon initialization, $\mathcal{F}_{\mathrm{sgx}}$ runs $\mathsf{outp} := \mathsf{prog}_{\mathrm{encl}}.\mathbf{Initialize}()$ 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 $\mathsf{prog}_{\mathrm{encl}}.\mathbf{Resume}(\mathsf{id}, \mathsf{params})$. Further formalism for $\mathcal{F}_{\mathrm{sgx}}$ is given in Appendix B.1.

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

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

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

Initialize:

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

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

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 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 within 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 12s and the clock serves in TC primarily to: (1) Schedule connections to data sources and (2) To check TLS certificates for expiration when establishing HTTPS connections. 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. Note that the trusted clock for SGX, backed by Intel Manageability Engine [23], is resilient to power outages and reboots [32].

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

5. TWO KEY SECURITY PROPERTIES

Before presenting the TC protocol details, we discuss two key security properties informing its design: gas sustainability and TCB minimization in TC's hybridized TCB model. While we introduce them in this work, as we shall explain, they are of broad and general applicability.

5.1 Gas Sustainability

As explained above, Ethereum's fee model requires that gas costs be paid by the user who initiates a transaction, including all costs resulting from dependent calls. This means that a service that initiates calls to Ethereum contracts must spend money to execute those calls. Without careful design, such services run the risk of malicious users (or protocol bugs) draining financial resources by triggering blockchain calls for which the service's fees will not be reimbursed. This could cause financial depletion and result in an application-layer denial-of-service attack. It is thus critical for the availability of Ethereum-based services that they always be reimbursed for blockchain computation they initiate.

To ensure that a service is not vulnerable to such attacks, we define *gas sustainability*, a new condition necessary for the liveness of blockchain contract-based services. Gas sustainability is a basic requirement for any self-perpetuating Ethereum service. It can also generalize beyond Ethereum; any decentralized blockchain-based smart contract system must require fees of some kind to reimburse miners for performing and verifying computation.

Let $\mathsf{bal}(\mathcal{W})$ denote the balance of an Ethereum wallet $\mathcal{W}$.

Definition 1 (K-Gas Sustainability)**.** *A service with wallet $\mathcal{W}$ and blockchain functions $f_1, \ldots, f_n$ is* K-gas sustainable *if the following holds. If $\mathsf{bal}(\mathcal{W}) \geq K$ prior to execution of any f_i and the service behaves honestly, then after each execution of an f_i initiated by $\mathcal{W}$, $\mathsf{bal}(\mathcal{W}) \geq K$.*

Recall that a call made in Ethereum with insufficient gas will abort, but spend all provided gas. While Ethereum trivially guarantees 0-gas sustainability, if a transaction is submitted by a wallet with insufficient funds, the wallet's balance will drop to 0. Therefore, to be K-gas sustainable for $K > 0$, each blockchain call made by the service must reimburse gas expenditures. Moreover, the service must have sufficient gas for each call or such reimbursement will be reverted with the rest of the transaction.

The need for gas sustainability (with $K > 0$, as required by TC) informs our protocol design in Section 6. We prove that TC achieves this property in Section 7.

5.2 Hybrid TCB Minimization

In a system involving a smart contract interacting with an off-chain trusted computing environment (e.g. SGX), the TCB is a hybrid of two components with distinct properties. Computation in the smart contract is slow, costly, and completely transparent, meaning it cannot rely on secrets. An SGX enclave is computationally powerful and executes privately, but all external interaction—notably including communication with the contract—must go through an untrusted intermediary. While this hybrid TCB is powerful and useful well beyond TC, it presents a challenge: establishing secure communication between the components while minimizing the code in the TCB.

We define abstractions for both TCB components in Fig. 4. To distinguish these abstractions from formal ideal functionalities, we use $\mathcal{T}$ (for trusted component), rather than $\mathcal{F}$. We

$\mathcal{T}_{\mathsf{Off}}$: abstraction for off-chain TCB

Initialize(void):
  $(\mathsf{pk}, \mathsf{sk}) := \Sigma.\mathsf{KeyGen}(1^\lambda)$
  Output pk

Resume(req):
  Assert $\mathcal{O}_{\mathsf{Auth}}(\mathsf{req})$
  $\mathsf{resp} := f(\mathsf{req})$
  $\sigma := \Sigma.\mathsf{Sign}(\mathsf{sk}, (\mathsf{req}, \mathsf{resp}))$
  Output $((\mathsf{req}, \mathsf{resp}), \sigma)$

$\mathcal{T}_{\mathsf{On}}$: abstraction for on-chain TCB

Request(req):
  Send (req) to $\mathcal{T}_{\mathsf{Off}}$

Deliver(req, resp, σ):
  $\Sigma.\mathsf{Verify}((\mathsf{req}, \mathsf{resp}), \sigma)$
  // can now use resp as trusted

Figure 4: Systems like TC have a hybrid TCB. Authentication between two components can greatly increase TCB complexity of implemented naively. We propose techniques to eliminate the most expensive operations (highlighted in red).

model the authentication of on-chain messages by an oracle $\mathcal{O}_{\mathsf{Auth}}$, which returns true if an input is a valid blockchain transaction. Since Ethereum blocks are self-authenticated using Merkle trees [14, 38], in principle we can realize $\mathcal{O}_{\mathsf{Auth}}$ by including an Ethereum client in the TCB. Doing so drastically increases the code footprint, however, as the core Ethereum implementation is about 50k lines of C++. Similarly, a smart contract could authenticate messages from an SGX by checking attestations, but implementing this verification in a smart contract would be error-prone and computationally (and thus financially) expensive.

Instead we propose two general techniques to avoid these calls and thereby minimize code size in the TCB. The first applies to any hybrid system where one TCB component is a blockchain contract. The second applies to any hybrid system where the TCB components communicate only to make and respond to requests.

Binding $\mathcal{T}_{\mathsf{Off}}$ to $\mathcal{W}_{TC}$. Due to the speed and cost of computation in the on-chain TCB, we wish to avoid implementing signature verification (e.g. Intel's EPID). There does exist a precompiled Ethereum contract to verify ECDSA signatures [38], but the operation requires a high gas cost. Instead, we describe here how to bind the identity of $\mathcal{T}_{\mathsf{Off}}$ to an Ethereum wallet, which allows $\mathcal{T}_{\mathsf{On}}$ to simply check the message sender, which is already verified as part of Ethereum's transaction protocol.

The key observation is that information can only be inserted into the Ethereum blockchain as a transaction from a wallet. Thus, the only way the Relay can relay messages from $\mathcal{T}_{\mathsf{Off}}$ to $\mathcal{T}_{\mathsf{On}}$ is through a wallet $\mathcal{W}_{TC}$. Since Ethereum itself already verifies signatures on transactions (i.e., users interact with Ethereum through an authenticated channel), we can piggyback verification of $\mathcal{T}_{\mathsf{Off}}$ signatures on top of the existing transaction signature verification mechanism. Simply put, the $\mathcal{T}_{\mathsf{Off}}$ creates $\mathcal{W}_{TC}$ with a fresh public key $\mathsf{pk}_{\mathsf{Off}}$ whose secret is known only to $\mathcal{T}_{\mathsf{Off}}$.

To make this idea work fully, the public key $\mathsf{pk}_{\mathsf{Off}}$ must be

hardcoded into $\mathcal{T}_{\mathsf{On}}$. A client creating or relying on a contract that uses $\mathcal{T}_{\mathsf{On}}$ is responsible for ensuring that this hardcoded $\mathsf{pk}_{\mathsf{Off}}$ has an appropriate SGX attestation before interacting with $\mathcal{T}_{\mathsf{On}}$. Letting Verify denote a verification algorithm for EPID signatures, Fig. 5 gives the protocol for a client to check that $\mathcal{T}_{\mathsf{On}}$ is backed by a valid $\mathcal{T}_{\mathsf{Off}}$ instance. (We omit the modeling here of IAS online revocation checks.)

In summary, by assuming that relying clients have verified an attestation of $\mathcal{T}_{\mathsf{Off}}$, we can assume that datagrams sent from $\mathcal{W}_{TC}$ are trusted to originate from $\mathcal{T}_{\mathsf{Off}}$. This eliminates the need to do costly EPID signature verification in $\mathcal{T}_{\mathsf{On}}$.

Additionally, SGX can seal $\mathsf{pk}_{\mathsf{Off}}$ in non-volatile storage while protecting integrity and confidentiality [8,22], allowing us to maintain the same binding through server restarts.

User: offline verification of SGX attestation

Inputs: $\mathsf{pk}_{\mathrm{sgx}}$, $\mathsf{pk}_{\mathsf{Off}}$, $\mathcal{T}_{\mathsf{Off}}$, σ_{att}

Verify:
Assert $\mathsf{pk}_{\mathsf{Off}}$ is the expected enclave code
Assert $\Sigma_{\mathrm{sgx}}.\mathsf{Verify}(\mathsf{pk}_{\mathrm{sgx}}, \sigma_{\mathrm{att}}, (\mathcal{T}_{\mathsf{Off}}, \mathsf{pk}_{\mathsf{Off}}))$
Assert $\mathcal{T}_{\mathsf{On}}$ is correct and parametrized with $\mathsf{pk}_{\mathsf{Off}}$
// now okay to rely on $\mathcal{T}_{On}$

Figure 5: A client checks an SGX attestation of the enclave's code $\mathcal{T}_{\mathsf{Off}}$ and public key $\mathsf{pk}_{\mathsf{Off}}$. The client also checks that $\mathsf{pk}_{\mathsf{Off}}$ is hardcoded into blockchain contract $\mathcal{T}_{\mathsf{On}}$ before using $\mathcal{T}_{\mathsf{On}}$.

Eliminating $\mathcal{O}_{\mathsf{Auth}}$. To eliminate the need to call $\mathcal{O}_{\mathsf{Auth}}$ from $\mathcal{T}_{\mathsf{Off}}$, we leverage the fact that all messages from $\mathcal{T}_{\mathsf{Off}}$ to $\mathcal{T}_{\mathsf{On}}$ are responses to existing requests. Instead of verifying request parameters in $\mathcal{T}_{\mathsf{Off}}$, we can verify in $\mathcal{T}_{\mathsf{On}}$ that $\mathcal{T}_{\mathsf{Off}}$ responded to the correct request. For each request, $\mathcal{T}_{\mathsf{On}}$ stores the parameters of that request. In each response, $\mathcal{T}_{\mathsf{Off}}$ includes the parameters it used to fulfill the request. $\mathcal{T}_{\mathsf{On}}$ can then check that the parameters in a response match the stored parameters and, if not, and simply reject. Storing parameters and checking equality are simple operations, so this vastly simpler than calling $\mathcal{O}_{\mathsf{Auth}}$ inside $\mathcal{T}_{\mathsf{Off}}$.

This approach may appear to open new attacks (e.g., the Relay can send bogus requests to which the $\mathcal{T}_{\mathsf{Off}}$ respond). As we prove in Section 7, however, all such attacks reduce to DoS attacks from the network or the Relay—attacks to which hybrid TCB systems are inherently susceptible and which we do not aim to protect against in TC.

6. TOWN CRIER PROTOCOL

We now present some preliminaries followed by the TC protocol. For simplicity, we assume a single instance of $\mathsf{prog}_{\mathrm{encl}}$, although our architecture could scale up to multiple enclaves and even multiple hosts.

To ensure gas sustainability, we require that requesters make gas payments up front as Ether. $\mathcal{C}_{TC}$ then reimburses the gas costs of TC. By having a trusted component perform the reimbursement, we are also able to guarantee that a malicious TC cannot steal an honest user's money without delivering valid data.

Notation. We use $\mathbf{msg}.m_i$ to label messages corresponding to those in Fig. 2. For payment, let $\$g$ 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, we assume that gas and currency adopt the same units (allowing us to avoid explicit conversions). We use the identifiers in Table 1 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
$\$\mathtt{G}_{\mathrm{min}}$	Gas required for **Deliver** excluding callback
$\$\mathtt{G}_{\mathrm{max}}$	Maximum gas TC can provide to invoke **Deliver**
$\$\mathtt{G}_{\mathrm{cncl}}$	Gas needed to invoke **Cancel**
$\$\mathtt{G}_{\emptyset}$	Gas needed for **Deliver** on a canceled request

Table 1: Identifiers for currency and gas amounts.

$\$\mathtt{G}_{\mathrm{min}}$, $\$\mathtt{G}_{\mathrm{max}}$, $\$\mathtt{G}_{\mathrm{cncl}}$, and $\$\mathtt{G}_{\emptyset}$ 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, a user-initiated transaction will abort if they are too small, so we need not worry about the values.

Initialization. TC deposits at least $\$\mathtt{G}_{\mathrm{max}}$ into the $\mathcal{W}_{TC}$.

The TC Contract $\mathcal{C}_{TC}$. The TC Contract accepts a datagram request with fee $\$f$ from $\mathcal{C}_U$, assigns it a unique id, and records it. The Town Crier Relay $\mathcal{R}$ monitors requests and forwards them to the Enclave. As we discussed in Section 5.2, upon receipt of a response from $\mathcal{W}_{TC}$, $\mathcal{C}_{TC}$ verifies that $\mathsf{params}' = \mathsf{params}$ to ensure validity. If the request is valid, $\mathcal{C}_{TC}$ forwards the resulting datagram data by calling the callback specified in the initial request. To ensure that all gas spent can be reimbursed, $\mathcal{C}_{TC}$ sets $\$g_{\mathrm{clbk}} := \$f - \$\mathtt{G}_{\mathrm{min}}$ for this sub-call. $\mathcal{C}_{TC}$ is specified fully in Fig. 6. Here, Call denotes a call to a contact entry point.

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 (id, params).
2. It boots the Enclave with $\mathsf{prog}_{\mathrm{encl}}$.**Initialize**() and calls $\mathsf{prog}_{\mathrm{encl}}$.**Resume**(id, params) on incoming requests.
3. It forwards datagrams from the Enclave 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 $\mathsf{prog}_{\mathrm{encl}}$.**Resume** exactly once with the parameters of each valid request and never otherwise.

The Enclave $\mathsf{prog}_{\mathrm{encl}}$. When initialized through **Initialize**(), $\mathsf{prog}_{\mathrm{encl}}$ 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 $\mathsf{prog}_{\mathrm{encl}}$ instance through insertion into attestations.

Upon a call to **Resume**(id, params), $\mathsf{prog}_{\mathrm{encl}}$ 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 $\mathsf{prog}_{\mathrm{encl}}$ fetches the

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

Initialize: Counter := 0

Request: On recv $(\mathsf{params}, \mathsf{callback}, \$\mathtt{f}, \$g_{\text{req}})$ from some $\mathcal{C}_U$:
Assert $\$\mathtt{G}_{\min} \le \$\mathtt{f} \le \$\mathtt{G}_{\max}$
id := Counter; Counter := Counter + 1
Store $(\mathsf{id}, \mathsf{params}, \mathsf{callback}, \$\mathtt{f}, \mathcal{C}_U)$ // $msg.m_1$
// $\$\mathtt{f}$ held by contract

Deliver: On recv $(\mathsf{id}, \mathsf{params}, \mathsf{data}, \$g_{\text{dvr}})$ from $\mathcal{W}_{TC}$:
(1) If isCanceled[id] and not isDelivered[id]
Set isDelivered[id]
(2) Send $\$\mathtt{G}_{\emptyset}$ to $\mathcal{W}_{TC}$
Return
Retrieve stored $(\mathsf{id}, \mathsf{params}', \mathsf{callback}, \$\mathtt{f}, \_)$
// abort if not found
Assert params = params′ and $\$\mathtt{f} \le \$g_{\text{dvr}}$
and isDelivered[id] not set
Set isDelievered[id]
(3) Send $\$\mathtt{f}$ to $\mathcal{W}_{TC}$
Set $\$g_{\text{clbk}} := \$\mathtt{f} - \$\mathtt{G}_{\min}$
(4) Call callback(data) with gas $\$g_{\text{clbk}}$ // $msg.m_4$

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

Figure 6: TC contract $\mathcal{C}_{TC}$ reflecting fees. The last argument of each function is the GASLIMIT provided.

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

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

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

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 $\mathcal{R}$.

requested datagram and returns it to $\mathcal{R}$ along with params, id, and a GASLIMIT $\$g_{\text{dvr}} := \$\mathtt{G}_{\max}$, all digitally signed with sk_{TC}. Fig. 8 shows the protocol for $\mathsf{prog}_{\text{encl}}$.

The Requester Contract $\mathcal{C}_U$. An honest requester first follows the protocol in Fig. 5 to verify the SGX attestation. Then she prepares params and callback, sets $\$g_{\text{req}}$ to the cost

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

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

Figure 8: The Town Crier Enclave $\mathsf{prog}_{\text{encl}}$.

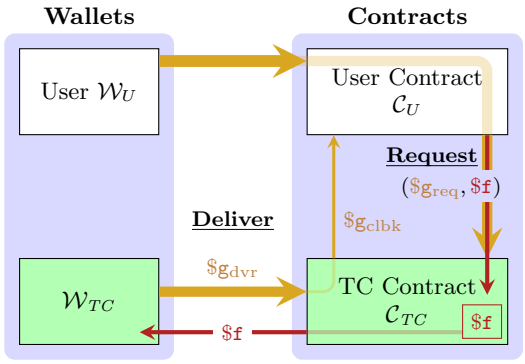

Figure 9: Money Flow for a Delivered Request. Red arrows denote flow of money and brown arrows denote gas limits. The thickness of lines indicate the quantity of resources. The $\$g_{\text{clbk}}$ arrow is thin because $\$g_{\text{clbk}}$ is limited to $\$\mathtt{f} - \$\mathtt{G}_{\min}$.

of **Request** with params, sets $\$\mathtt{f}$ to $\$\mathtt{G}_{\min}$ plus the cost of executing callback, and invokes **Request**(params, callback, $\$\mathtt{f}$) with GASLIMIT $\$g_{\text{req}}$.

If callback is not executed, she can invoke **Cancel**(id) with GASLIMIT $\$\mathtt{G}_{\text{cncl}}$ to receive a partial refund. An honest requester will invoke **Cancel** at most once for each of her requests and never for any other user's request.

6.1 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 8.1.

6.2 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 [18,39] 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 8.2. (A potential optimization is mentioned in Section 10.)

6.3 Implementation Details

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

7. 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 2 (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 (prog_{encl}.**Resume**$(\mathsf{id}, \mathsf{params})$ in our model). More formally, for any probabilistic polynomial-time adversary $\mathcal{A}$,*

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

for security parameter λ.

Theorem 1 (Authenticity)**.** *Assume that Σ_{sgx} and Σ are secure signature schemes. Then, the TC protocol achieves authenticity of data feed under Definition 2.*[3]

Fee Safety. Our protocol in Section 6 ensures that an honest Town Crier will not run out of money and that an honest requester will not pay excessive fees.

Theorem 2 (Gas Sustainability)**.** *Town Crier is* $\$\mathtt{G}_{\max}$-gas sustainable.

[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 [35].

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 $\mathsf{callback}$. 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* $\mathsf{callback}$*, let* $\$\mathtt{G}_{req}$ *and* $\$\mathtt{F}$ *be the honestly-chosen values of* $\$\mathtt{g}_{req}$ *and* $\$\mathtt{f}$*, respectively, when submitting the request* $(\mathsf{params}, \mathsf{callback}, \$\mathtt{f}, \$\mathtt{g}_{req})$*. For any such request submitted by an honest user, one of the following holds:*

- $\mathsf{callback}$ *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 6.2, 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 8.1. 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.

8. EXPERIMENTS

We implemented three showcase applications which we plan to launch together with TC. We provide a brief description of our applications followed by cost and performance measurements. We refer the reader to Appendix D for more details on the applications and code samples.

8.1 Requesting Contracts

Financial Derivative ($\mathsf{CashSettledPut}$)**.** 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 $\mathsf{CashSettledPut}$ 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 ($\mathsf{FlightIns}$)**.** Flight insurance indemnifies a purchaser should her flight be delayed or canceled. We have implemented a simple flight insurance contract called $\mathsf{FlightIns}$. 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 delayed or 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 [4], 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 the Enclave can log in as the seller and determine from the web-page whether the virtual item has been shipped.

8.2 Measurements

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.

We report mean run times (with the standard deviation in parenthesis) over 100 trials.

TCB Size. The trusted computing base (TCB) of Town Crier includes the Enclave and TC Contract. The Enclave consists of approximately 46.4k lines of C/C++ code, the vast majority of which (42.7k lines) is the modified mbedTLS library [9]. The source code of mbedTLS has been widely deployed and tested, while the remainder of the Enclave codebase is small enough to admit formal verification. The TC Contract is also compact; it consists of approximately 120 lines of Solidity code.

Enclave Response Time. We measured the enclave response time for handling a TC request, defined as the interval between (1) the Relay sending a request to the enclave and (2) the Relay receiving a response from the enclave.

Table 2 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 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.

Transaction Throughput. 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. Fig. 10 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 under 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. Recent work [19] indicates that Bitcoin cannot scale beyond 26 tx/sec without a protocol redesign. Thus, with few hosts TC can easily meet the data feed demands of even future decentralized blockchains.

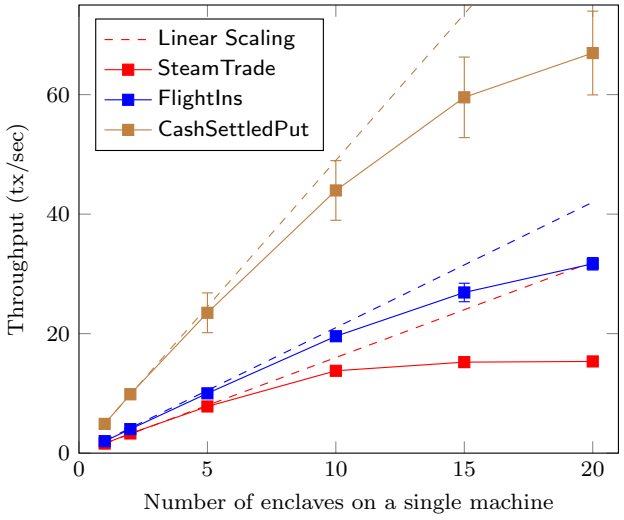

Figure 10: Throughput on a single SGX machine. The x-axis is the number of concurrent enclaves and the y-axis is the number of tx/sec. Dashed lines indicate the ideal scaling for each application, and error bars, the standard deviation. We ran 20 rounds of experiments (each round processing 1000 transactions in parallel).

Gas Costs. Currently 1 gas costs 5×10^{-8} Ether, so at the exchange rate of $15 per Ether, $1 buys 1.3 million gas. Here we provide costs for our implementation components.

The callback-independent portion of **Deliver** costs about 35,000 gas (2.6¢), so this is the value of $\$G_{\min}$. We set $\$G_{\max}$ = 3,100,000 gas ($2.33), as this is approximately Ethereum's maximum `GASLIMIT`. The cost for executing **Request** is approximately 120,000 gas (9¢) of fixed cost, plus 2500 gas (0.19¢) for every 32 bytes of request parameters. The cost to execute **Cancel** is 62500 gas (4.7¢) including the gas cost $\$G_{\text{cncl}}$ and the refund $\$G_{\emptyset}$ paid to TC should **Deliver** be called after **Cancel**.

The total callback-independent cost of acquiring a datagram from TC (i.e., the cost of the datagram, not the application) ranges from 11.9¢ (CashSettledPut) to 12.9¢ (SteamTrade)[4]. The variation results from differing parameter lengths.

Component-Compromise Resilience. For the CashSettledPut application, we implemented and evaluated two modes of majority voting (as in Section 6.2):

- 2-out-of-3 majority voting within the enclave, providing 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 en-

[4]This cost is for 1 item. Each additional item costs 0.19¢.

	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 2: 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 *Enclave Response Time*. Times may not sum to this total due to minor unprofiled overhead.

clave 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). 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. In this scenario the gas cost would increase by a factor of 3 plus an additional 5.85¢. So CashSettledPut would cost 35.6¢ for Deliver without Cancel. The extra 5.85¢ is the cost to store votes until a winner is known.

Offline Measurements. Recall that an enclave requires a one-time setup operation that involves attestation generation. 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 the report, and 54.9 (10.3) ms for the quote.

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 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, rarely more than a few hundred milliseconds.

9. RELATED WORK

Virtual Notary [6, 27] is an early online data attestation service that verifies and digitally signs any of a range of user-requested "factoids" (web page contents, stock prices, etc.) potentially suitable for smart contracts. It predates and does not at present interface with Ethereum.

Several data feeds are deployed today for smart contract systems such as Ethereum. Examples include PriceFeed [3] and Oraclize.it [7]. The latter achieves distributed trust by using a second service called TLSnotary [5], which digitally signs TLS session data. As a result, unlike TC which can flexibly tailor datagrams, Oraclize.it must serve data verbatim from a web session or API call; verbose sources thus mean superfluous data and inflated gas costs. Additionally, these services ultimately rely on the reputations of their (small) providers to ensure data authenticity and cannot support private or custom datagrams. Alternative systems such as SchellingCoin [13] and Augur [2] rely on prediction markets to decentralize trust, creating a heavy reliance on human input and severely constraining their scope and data types.

Despite an active developer community, research results on smart contracts are limited. Work includes off-chain contract execution for confidentiality [28], and, more tangentially, exploration of e.g., randomness sources in [11]. The only research involving data feeds to date explores criminal applications [26].

SGX is similarly in its infancy. While a Windows SDK [24] and programming manual [22] have just been released, a number of pre-release papers have already explored SGX, e.g., [8, 29, 31, 33, 39]. Researchers have demonstrated applications including enclave execution of legacy (non-SGX) code [10] and use of SGX in a distributed setting for map-reduce computations [33]. Several works have exposed shortcomings of the security model for SGX [18,34,35], including side-channel attacks against enclave state.

10. FUTURE WORK

We plan to develop TC after its initial deployment to incorporate a number of additional features. We discuss a few of those features here.

Freeloading Protection. There are concerns in the Ethereum community about "parasite contracts" that forward or resell datagrams from fee-based data feeds [37]. As a countermeasure, we plan to deploy the following mechanism in TC inspired by designated verifier proofs [25]. The set of n users $\mathcal{U} = \{U_1, \ldots, U_n\}$ of a requesting contract generate an (n, n)-secret-shared key pair $(\mathsf{sk}_{\mathcal{U}}, \mathsf{pk}_{\mathcal{U}})$. They submit their n individual shares to the TC Enclave (e.g., as ciphertexts under pk_{TC} sent to $\mathcal{C}_{TC}$).

TC now can sign datagrams using $\mathsf{sk}_{\mathcal{U}}$. Each user U_i can be sure individually that a datagram produced by TC is valid, since she did not collude in its creation. Potential parasitic users, however, cannot determine whether the datagram was produced by $\mathcal{C}_{TC}$ or by $\mathcal{U}$, and thus whether or not it is valid. Such a *source-equivocal datagram* renders parasite contracts less trustworthy and thus less attractive.

Revocation Support. There are two forms of revocation relevant to TC. First, the certificates of data sources may be revoked. Since TC already uses HTTPS, it could easily use the Online Certificate Status Protocol (OCSP) to check TLS certificates. Second, an SGX host could become compromised, prompting revocation of its EPID signatures by Intel. The Intel Attestation Service (IAS) will reportedly disseminate such revocations. Conveniently, clients already use

the IAS when checking the attestation σ_{att}, so revocation checking will require no modification to TC.

Hedging Against SGX Compromise. We discussed in Section 6.2 how TC can support majority voting across SGX hosts and data sources. Design enhancements to TC could reduce associated latency and gas costs. For SGX voting, we plan to investigate a scheme in which SGX-enabled TC hosts agree on a datagram value X via Byzantine consensus. The hosts may then use a threshold digital signature scheme to sign the datagram response from $\mathcal{W}_{TC}$, and each participating host can monitor the blockchain to ensure delivery.

Updating TC's Code. As with any software, we may discover flaws in TC or wish to add new functionality after initial deployment. With TC as described above, however, updating $\mathsf{prog}_{\text{encl}}$ would cause the Enclave to lose access to sk_{TC} and thus be unable to respond to requests in $\mathcal{C}_{TC}$. The TC operators could set up a new contract $\mathcal{C}'_{TC}$ referencing new keys, but this would be expensive and burdensome for TC's operators and users. While arbitrary code changes would be insecure, we could create a template for user contracts that includes a means to approve upgrades. We plan to investigate this and other mechanisms.

Generalized Custom Datagrams. In our SteamTrade example contract we demonstrated a custom datagram that scrapes a user's online account using her credentials. A more generic approach would allow users to supply their own general-purpose code to TC and achieve inexpensive data-source-enriched emulation of private contracts in Hawk [28]. Placing such large requests on the blockchain would be prohibitively expensive, but code could be easily loaded into TC enclave off-chain. Of course, deploying arbitrary user code raises many security and confidentiality concerns which TC would need to address.

11. CONCLUSION

We have introduced Town Crier (TC), an authenticated data feed for smart contracts specifically designed to support Ethereum. Use of Intel's new SGX trusted hardware allows TC to serve datagrams with a high degree of trustworthiness. We defined *gas sustainability*, a critical availability property of Ethereum services, and provided techniques for shrinking the size of a hybrid TCB spanning the blockchain and an SGX. We proved in a formal model that TC serves only data from authentic sources, and showed that TC is gas sustainable and minimizes cost to honest users should the code behave maliciously. In experiments involving end-to-end use of the system with the Ethereum blockchain, we demonstrated TC's practicality, cost effectiveness, and flexibility for three example applications. We believe that TC offers a powerful, practical means to address the lack of trustworthy data feeds hampering Ethereum evolution today and that it will support a rich range of applications. Pending deployment of the Intel Attestation Service (IAS), we will make a version of TC freely available as a public service.

Acknowledgements

This work is funded in part by NSF grants CNS-1314857, CNS-1330599, CNS-1453634, CNS-1518765, CNS-1514261, a Packard Fellowship, a Sloan Fellowship, Google Faculty Research Awards, and a VMWare Research Award. Our thanks also to Andrew Miller and Gun Sirer for their very helpful insights and comments on this work.

12. REFERENCES

[1] http://coinmarketcap.com/currencies/ethereum.
[2] Augur. http://www.augur.net/.
[3] PriceFeed smart contract. Referenced Feb. 2016 at http://feed.ether.camp/.
[4] Steam online gaming platform. http://store.steampowered.com/.
[5] TLSnotary – a mechanism for independently audited https sessions. https://tlsnotary.org/TLSNotary.pdf, 10 Sept. 2014.
[6] Cornell researchers unveil a virtual notary. Slashdot, 20 June 2013.
[7] Oraclize: "The provably honest oracle service". www.oraclize.it, Referenced Feb. 2016.
[8] I. Anati, S. Gueron, and S. Johnson. 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.
[9] ARM Limited. mbedTLS (formerly known as PolarSSL). https://tls.mbed.org/.
[10] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*, 2014.
[11] J. Bonneau, J. Clark, and S. Goldfeder. On bitcoin as a public randomness source. https://eprint.iacr.org/2015/1015.pdf, 2015.
[12] E. Brickell and J. Li. Enhanced Privacy ID from Bilinear Pairing. *IACR Cryptology ePrint Archive*, 2009:95, 2009.
[13] V. Buterin. Schellingcoin: A minimal-trust universal data feed. https://blog.ethereum.org/2014/03/28/schellingcoin-a-minimal-trust-universal-data-feed/.
[14] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.
[15] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.
[16] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *Theory of Cryptography*, pages 61–85. Springer, 2007.
[17] R. Canetti and T. Rabin. Universal composition with joint state. In *CRYPTO*, 2003.
[18] V. Costan and S. Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org/.
[19] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer. On scaling decentralized blockchains (a position paper). In *Bitcoin Workshop*, 2016.
[20] Ethereum. go-ethereum. https://github.com/ethereum/go-ethereum.
[21] G. Greenspan. Why many smart contract use cases are simply impossible. http://www.coindesk.com/three-smart-contract-misconceptions/.

[22] Intel Corporation. *Intel® Software Guard Extensions Programming Reference*, 329298-002us edition, 2014.
[23] Intel Corporation. Intel® Software Guard Extensions Evaluation SDK User's Guide for Windows* OS. https://software.intel.com/sites/products/sgx-sdk-users-guide-windows, 2015.
[24] Intel Corporation. Intel® Software Guard Extensions SDK. https://software.intel.com/en-us/sgx-sdk, 2015.
[25] M. Jakobsson, K. Sako, and R. Impagliazzo. Designated verifier proofs and their applications. In *Advances in Cryptology – EUROCRYPT '96*, pages 143–154. Springer, 2001.
[26] A. Juels, A. Kosba, and E. Shi. The Ring of Gyges: Investigating the future of criminal smart contracts. Online manuscript, 2015.
[27] A. Kelkar, J. Bernard, S. Joshi, S. Premkumar, and E. G. Sirer. Virtual Notary. http://virtual-notary.org/, 2016.
[28] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy*, 2016.
[29] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.
[30] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
[31] V. Phegade and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–1, New York, New York, USA, 2013. ACM Press.
[32] X. Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 2014.
[33] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud. In *IEEE S&P*, 2015.
[34] E. Shi. Trusted hardware: Life, the composable university, and everything. Talk at the DIMACS Workshop on Cryptography and Big Data, 2015.
[35] E. Shi, F. Zhang, R. Pass, S. Devadas, D. Song, and C. Liu. Trusted hardware: Life, the composable universe, and everything. Manuscript, 2015.
[36] N. Szabo. Smart contracts. http://szabo.best.vwh.net/smart.contracts.html, 1994.
[37] K. Torpey. The conceptual godfather of augur thinks the project will fail. CoinGecko, 5 Aug. 2015.
[38] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.
[39] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656, May 2015.

APPENDIX

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 $\mathcal{C}_{TC}$ as described in Section 6 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 [22]. 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. 8 shows the Enclave code $\mathsf{prog}_{\mathrm{encl}}$. 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. 11.

The Enclave. There are three components to the enclave

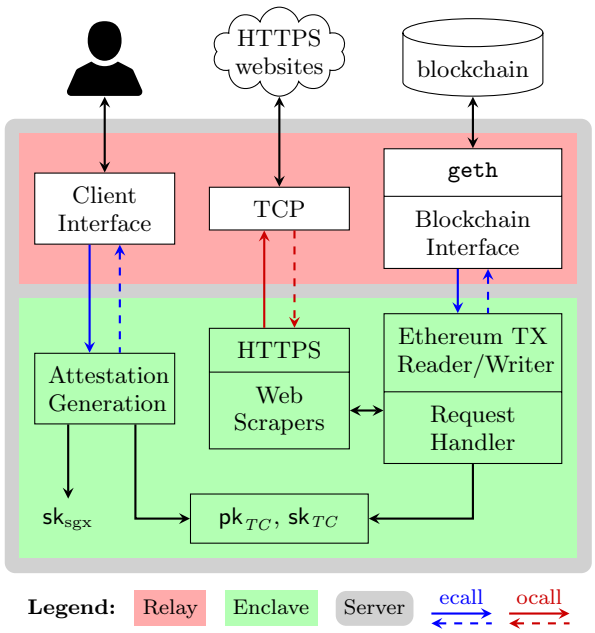

Figure 11: Components of TC Server

code $\mathsf{prog}_{\mathrm{encl}}$: 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` [9]) 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 Section 10.)

Web Scrapers. We implemented scrapers for our examples in Section 8.1 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 $\mathsf{prog}_{\mathrm{encl}}$. 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 [20]. 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. FORMAL MODELING

B.1 SGX Formal Modeling

As mentioned earlier, we adopt the UC model of SGX proposed by Shi et al. [35] In particular, their abstraction captures a subset of the features of Intel SGX. The main idea behind the UC modeling by Shi et al. [35] is to think of SGX as a trusted third party defined by a global functionality $\mathcal{F}_{\mathrm{sgx}}$ (see Figure 3 of Section 4.3).

Modeling choices. For simplicity, the $\mathcal{F}_{\mathrm{sgx}}$ 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 $\mathsf{pk}_{\mathrm{sgx}}$ and $\mathsf{sk}_{\mathrm{sgx}}$ (i.e., think of always signing with the 0-th key of the group signature scheme). We adopt this notational choice from [35] 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. [35] 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 $\mathsf{outp} := \mathsf{prog}_{\text{encl}}.\textbf{Initialize}()$. 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 $\mathsf{prog}_{\text{encl}}.\textbf{Resume}$ on the input parameters denoted params. $\mathcal{F}_{\text{sgx}}$ outputs whatever $\mathsf{prog}_{\text{encl}}.\textbf{Resume}$ 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 [35] 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 [35] 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. [28]. In addition to UC modeling of blockchain-based protocols, Kosba et al. [28] 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 [28] 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.

As we will see in the proof of Theorem 2, we must carefully handle the case of **Deliver** arriving after **Cancel**. In the formal blockchain model proposed by Kosba et al. [28], 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 do 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 [28]. 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 $\$\mathtt{G}_{\max}$-gas sustainability 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 7.

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 2.*

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 was 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 Sustainability)**.** *Town Crier is* $\$\mathtt{G}_{\max}$-gas sustainable.

Proof. By assumption, $\mathcal{W}_{TC}$ is seeded with at least $\$\mathtt{G}_{\max}$ 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 params = params′. The enclave always provides $\$\mathtt{g}_{\text{dvr}} = \$\mathtt{G}_{\max}$ (which it has by assumption) and **Request** ensures that $\$\mathtt{f} \leq \$\mathtt{G}_{\max}$. 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 $\$\mathtt{G}_{\min}$ 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 $\$\mathtt{f} - \$\mathtt{G}_{\min}$ and line (4) is limited to using $\$\mathtt{g}_{\text{clbk}}$ gas, the total gas spent on this execution of **Deliver** is at most $\$\mathtt{G}_{\min} + (\$\mathtt{f} - \$\mathtt{G}_{\min}) = \$\mathtt{f}$.
- 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 $\$\mathtt{G}_{\min}$ 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 $\$\mathtt{G}_{\min}$ 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* (params, 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 first spend $\$\mathtt{G}_{\text{req}}$ to invoke **Request**(params, 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 params = 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, $\$\mathtt{F} - \$\mathtt{G}_{\min}$ 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

$$\$\mathtt{G}_{\text{req}} + \$\mathtt{G}_{\text{cncl}} + \$\mathtt{F} - (\$\mathtt{F} - \$\mathtt{G}_{\emptyset}) = \$\mathtt{G}_{\text{req}} + \$\mathtt{G}_{\text{cncl}} + \$\mathtt{G}_{\emptyset}.$$

□

D. APPLICATIONS AND CODE SAMPLES

We now elaborate on the applications described in Section 8.1 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 12 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
$\$\mathtt{F}_{\text{TC}}$:= fee payed to TC for datagram delivery

Functions

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

Buy: On recv $(X, \$\mathtt{f})$ from $\mathcal{W}_U$:
Assert isRecovered not set
and timestamp $<$ expr
and $\mathcal{W}_{\text{buyer}}$ not set
and $X \leq M$
and $\$\mathtt{f} = (X \cdot P_U)$
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 $P_S \cdot X + \$\mathtt{F}_{\text{TC}}$

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

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 12: 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 13 we include a full implementation of FlightIns in Solidity.

Steam Marketplace (SteamTrade). Steam [4] 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 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 14 we describe the protocol for SteamTrade. We again omit the full source code due to length and complexity.

```
// 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 13: Solidity code for the FlightIns application contract.

SteamTrade **blockchain contract**

Constants

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

Functions

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

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

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

Figure 14: The FlightIns **application contract**