TriaticsDAO

VOYAGER ORACLE

TECHNICAL WHITEPAPER

1 INTRODUCTION

Smart contracts are applications that execute on decentralized infrastructure, such as a blockchain. They are tamperproof, in the sense that no party (even their creator) can alter their code or interfere with their execution. Historically, contracts embodied in code have run in a centralized manner that leaves them subject to alteration, termination, and even deletion by a privileged party. In contrast, smart contracts' execution guarantees, which bind all parties to an agreement as written, create a new and powerful type of trust relationship that does not rely on trust in any one party. Because they are self-verifying and self-executing, smart contracts offer a superior vehicle for realizing and administering digital agreements.

The powerful new trust model that smart contracts embody, though, introduces a new technical challenge: connectivity. The vast majority of interesting [27] smart contract applications rely on data about the real world that comes from key resources,
specifically, data feeds and APIs, that are external to the blockchain. Because of the mechanics of the consensus mechanisms underpinning blockchains, a blockchain cannot directly fetch such critical data.

We propose a solution to the smart contract connectivity problem in the form of Triatics, a secure oracle network. What differentiates Triatics from other oracle solutions is its ability to operate as a fully decentralized network. This decentralized approach limits the trust in any single party, enabling the tamperproof quality valued in smart contracts to be extended to the end-to-end operation between smart contracts and the APIs they rely on. Making smart contracts externally aware, meaning capable of interacting with off-chain resources, is necessary if they are going to replace the digital agreements in use today.
Today, the lion's share of traditional contractual agreements that have been digitally automated use external data to prove contractual performance, and require data outputs to be pushed to external systems. When smart contracts replace these older contractual mechanisms, they will require high-assurance versions of the same types of data inputs and outputs. Examples of potential next-generation smart contracts and their data requirements include:

- Securities smart contracts such as bonds, interest rate derivatives, and many others will require access to APIs reporting market prices and market reference data, e.g. interest rates.

- Insurance smart contracts will need data feeds about IoT data related to the insurable event in question, e.g.: was the warehouse's magnetic door locked at the time of breach, was the company's firewall online, or did the flight you had insurance for arrive on time.

- Trade finance smart contracts will need GPS data about shipments, data from supply chain ERP systems, and customs data about the goods being shipped in order to confirm fulfillment of contractual obligations

The main use of smart contracts in Ethereum today is management of tokens, which are a common functionality in most smart contract networks. We believe that the current focus on tokens to the exclusion of many other possible applications is due to a lack of adequate oracle services, a situation Triatics specifically aims to remedy.

Another problem common to these examples is the inability for smart contracts to output data into off-chain systems. Such output often takes the form of a payment message routed to traditional centralized infrastructure in which users already have accounts, e.g., for bank payments, PayPal, and other payment networks. Triatic's ability to securely push data to APIs and various legacy systems on behalf of a smart contract permits the creation of externally-aware tamperproof contracts.

2 ARCHITECTURAL OVERVIEWS

Triatic's core functional objective is to bridge two environments: on-chain and off-chain. We describe the architecture of each Triatics component below. Triatics will initially be built on Ethereum [16], [35], but we intend for it to support all leading smart contract networks for both off-chain and cross-chain interactions. In both its on and off-chain versions, Triatics has been designed with modularity in mind. Every piece of the Triatics system is upgradable, so that different components can be replaced as better techniques and competing implementations arise.

2.1 On-Chain Architecture

As an oracle service, Triatics nodes return replies to data requests or queries made by or on behalf of a user contract, which we refer to as requesting contracts and denote by USER-SC. Triatics' on-chain interface to requesting contracts is itself an on-chain contract that we denote by TRIATICS-SC.

Behind TRIATICS-SC, Triatics has an on-chain component consisting of three main contracts: a reputation contract, an order-matching contract, and an aggregating contract. The reputation contract keeps track of oracle-service-provider performance metrics. The order-matching smart contract takes a proposed service level agreement, logs the SLA parameters, and collects bids from oracle providers. It then selects bids using the reputation contract and finalizes the oracle SLA. The aggregating contract collects the oracle providers' responses and calculates the final collective result of the Triatics query. It also feeds oracle provider metrics back into the reputation contract. Triatics contracts are designed in a modular manner, allowing for them to be configured or replaced by users as needed. The on-chain work flow has three steps: 1) oracle selection, 2) data reporting, 3) result aggregation.

Oracle Selection

An oracle services purchaser specifies requirements that make up a service level agreement (SLA) proposal. The SLA proposal includes details such as query parameters and the number of oracles needed by the purchaser. Additionally, the purchaser specifies the reputation and aggregating contracts to be used for the rest of the agreement.

Using the reputation maintained on-chain, along with a more robust set of data gathered from logs of past contracts, purchasers can manually sort, filter, and select oracles via off-chain listing services. Our intention is for Triatics to maintain one such listing service, collecting all Triatics-related logs and verifying the binaries of listed oracle contracts. We further detail the listing service and reputation systems in Section 5. The data used to generate listings will be pulled from the blockchain, allowing for alternative oracle-listing services to be built. Purchasers will submit SLA proposals to oracles off-chain, and come to agreement before finalizing the SLA on-chain.

Manual matching is not possible for all situations. For example, a contract may need to request oracle services dynamically in response to its load. Automated solutions solve this problem and enhance usability. For these reasons, automated oracle matching is also being proposed by Triatics through the use of order matching contracts.

Once the purchaser has specified their SLA proposal, instead of contacting the oracles directly, they will submit the SLA to an order-matching contract. The submission of the proposal to the order-matching contract triggers a log that oracle providers can monitor and filter based on their capabilities and service objectives. Triatics nodes then choose whether to bid on the proposal or not, with the contract only accepting bids from nodes that meet the SLA's requirements. When an oracle service provider bids on a contract, they commit to it, specifically by attaching the penalty amount that would be lost due to their misbehavior, as defined in the SLA.

Bids are accepted for the entirety of the bidding window. Once the SLA has received enough qualified bids and the bidding window has ended, the requested number of oracles is selected from the pool of bids. Penalty payments that were offered during the bidding process are returned to

oracles who were not selected, and a finalized SLA record is created. When the finalized SLA is recorded it triggers a log notifying the selected oracles. The oracles then perform the assignment detailed by the SLA.

Data Reporting

Once the new oracle record has been created, the off-chain oracles execute the agreement and report back on-chain. For more detail about off-chain interactions, see Sections 2.2 and 4.

Result Aggregation

Once the oracles have revealed their results to the oracle con- tract, their results will be fed to the aggregating contract. The aggregating contract tallies the collective results and calculates a weighted answer. The validity of each oracle response is then reported to the reputation contract. Finally, the weighted answer is returned to the specified contract function in USER-SC.

Detecting outlying or incorrect values is a problem that is specific to each type of data feed and application. For instance, detecting and rejecting outlying answers before averaging may be necessary for numeric data but not Boolean. For this reason, there will not be a specific aggregating contract, but a configurable contract address which is specified by the purchaser. Triatics will include a standard set of aggregating contracts, but customized contracts may also be specified, provided they conform to the standard calculation interface.

2.2 Off-Chain Architecture

Off-chain, Triatics initially consists of a network of oracle nodes connected to the Ethereum network, and we intend for it to support all leading smart contract networks. These nodes independently harvest responses to off-chain requests. As we explain below, their individual responses are aggregated via one of several possible consensus mechanisms into a global response that is returned to a requesting con- tract USER-SC. The Triatics nodes are powered by the standard open source core implementation which handles standard blockchain interactions, scheduling, and connecting with common external resources. Node operators may choose to add software

extensions, known as external adapters, that allow the operators to offer additional specialized off-chain services. Triatics nodes have already been deployed along- side both public blockchains and private networks in enterprise settings; enabling the nodes to run in a decentralized manner is the motivation for the Triatics network.

Triatics Core

The core node software is responsible for interfacing with the blockchain, scheduling, and balancing work across its various external services. Work done by Triatics nodes is formatted as assignments. Each assignment is a set of smaller job specifications, known as subtasks, which are processed as a pipeline. Each subtask has a specific operation it performs, before passing its result onto the next subtask, and ultimately reaching a final result. Triatics' node software comes with a few subtasks built in, including HTTP requests, JSON parsing, and conversion to various blockchain formats.

External Adapters

Beyond the built-in subtask types, custom subtasks can be defined by creating adapters. Adapters are external services with a minimal REST API. By modeling adapters in a service-oriented manner, programs in any programming language can be easily implemented simply by adding a small intermediate API in front of the program. Similarly, interacting with complicated multi-step APIs can be simplified to individual subtasks with parameters.

Subtask Schemas

We anticipate that many adapters will be open sourced, so that services can be audited and run by various community members. With many different types of adapters being developed by many different developers, ensuring compatibility between adapters is essential.

Triatics currently operates with a schema system based on JSON Schema [36], to specify what inputs each adapter needs and how they should be formatted. Similarly, adapters specify an output schema to describe the format of each subtask's output.

3 Oracle Security

In order to explain Triatics's security architecture, we must first explain why security is important—and what it means.

Why must oracles be secure? Returning to our examples in Section 1, if a smart contract security gets a false data feed, it may payout the incorrect party. If smart contract insurance data feeds can be tampered with by the insured party there may be insurance fraud, and if GPS data given to a trade finance contract can be modified after it leaves the data provider, payment can be released for goods that haven't arrived.

More generally, a well-functioning blockchain, with its ledger or bulletin-board abstraction, offers very strong security properties. Users rely on the blockchain as a functionality that correctly validates transactions and prevents data from being altered. They treat it in effect like a trusted third party (a concept we discuss at length below). A supporting oracle service must offer a level of security commensurate with that of the blockchain it supports. An oracle too must therefore serve users as an effective trusted third party, providing correct and timely responses with very high probability. The security of any system is only as strong as its weakest link, so a highly trustworthy oracle is required to preserve the trustworthiness of a well-engineered blockchain.

Defining oracle security: An ideal view

In order to reason about oracle security, we must first define it. An instructive, principled way to reason about oracle security stems from the following thought experiment. Imagine that a trusted third party (TTP)—an ideal entity or functionality that always carries out instructions faithfully to the letter—were tasked with running an oracle. We'll denote this oracle by ORACLE (using all caps in general to denote an entity fully trusted by users), and suppose that the TTP obtains data from a perfectly trustworthy data source SRC. Given this magical service ORACLE, what instructions would we ask it to carry out? To achieve the property of integrity, also referred to as the authenticity property [24], we would simply ask that ORACLE perform the following steps:

1. Accept request: Ingest from a smart contract USER-SC a request Req = (Src, τ, q) that specifies a target data source Src, a time or range of times τ , and a query q;
2. Obtain data: Send query q to Src at time τ;
3. Return data: On receiving answer a, return a to the smart contract

These simple instructions, correctly carried out, define a strong, meaningful, but simple notion of security. Intuitively, they dictate that ORACLE acts as a trustworthy bridge between Src and USER-SC.

Confidentiality is another desirable property for oracles. As USER-SC sends Req to ORACLE in the clear on the blockchain, Req is public. There are many situations in which Req is sensitive and its publication could be harmful. If USER-SC is a flight insurance contract, for example, and sends ORACLE a query Req regarding a particular user's flight (q = "Ether Air Flight 338"), the result would be that a user's flight plans are revealed to the whole world. If USER-SC is a contract financial trading, Req could leak information about a user's trades and portfolio. There are many other examples, of course.

To protect the confidentiality of Req, we can require that data in Req be encrypted under a (public key) belonging to ORACLE. Continuing to leverage the TTP nature of ORACLE, we could then simply give ORACLE the information-flow constraint:

Upon decrypting Req, never reveal or use data in Req except to query Src

There are other important oracle properties, such as availability, the last of the classical CIA (Confidentiality-Integrity-Availability) triad. A truly ideal service ORACLE, of course, would never go down. Availability also encompasses more subtle properties such as censorship resistance: An honest ORACLE will not single out particular smart contracts and deny their requests.

The concept of a trusted third party is similar to the notion of an ideal functionality [7] used to prove the security of cryptographic protocols in certain models. We can also model a blockchain in similar terms,

conceptualizing it in terms of a TTP that maintains an ideal bulletin board. Its instructions are to accept transactions, validate them, serialize them, and maintain them permanently on the bulletin board, an append-only data structure.

4 TRIATICS Decentralization Approach

We propose three basic complementary approaches to ensuring against faulty nodes:

(1) Distribution of data sources; (2) Distribution of oracles; and (3) Use of trusted hardware.

We discuss the first two approaches, which involve decentralization, in this section. We discuss our long-term strategy for trusted hardware, a different and complementary approach, in Section 6.

4.1 Distributing sources

A simple way to deal with a faulty single source Src is to obtain data from multiple sources, i.e., distribute the data source. A trustworthy ORACLE can query a collection of sources Src1, Src2, . . . , Srck, obtain responses a1, a2, . . . , ak, and aggregate them into a single answer A = agg(a1, a2, . . . , ak).

ORACLE might do this in any of a number of ways. One, for example, is majority voting. If a majority of sources return the identical value a, the function agg returns a; otherwise it returns an error. In this case, provided that a majority (> k/2) sources are functioning correctly, ORACLE will always return a correct value A.

Many alternative functions agg can ensure robustness against erroneous data or handle fluctuations in data values over time (e.g, stock prices). For example, agg might discard outliers (e.g., the largest and smallest values ai) and output the mean of the remaining ones.

Of course, faults may be correlated across data sources in a way that weakens the assurances provided by aggregation. If site Src1 = EchEch.com obtains its data from Src2 = TheMoth.com, an error at Src2 will always imply an error at Src1. More subtle correlations between data sources can also occur.

Triatics also proposes to pursue research into mapping and reporting the independence of data sources in an easily digestible way so that oracles and users can avoid undesired correlations.

4.2 Distributing oracles

Just as sources can be distributed, our ideal service ORACLE itself can be approximated as a distributed system. This is to say that instead of a single monolithic oracle node O, we can instead have a collection of n different oracle nodes O1,O 2, . . . ,O n . Each oracle Oi contacts its own distinct set of data sources which may or may not overlap with those of other oracles. Oi aggregates responses from its data sources and outputs its own distinct answer Ai to a query Req.

Some of these oracles may be faulty. So clearly the set of all oracles' answers A1, A2, . . . , An will need to be aggregated in a trustworthy way into a single, author- itative value A.

Off-chain aggregation

This approach aggregates oracle responses off-chain and transmit a single message to TRIATICS-SC A. We propose deployment of this approach, called off-chain aggregation, in the medium-to-long term.

The problem of achieving a consensus value A in the face of potentially faulty nodes is much like the problem of consensus that underpins blockchains themselves. Given a predetermined set of oracles, one might consider using a classical Byzantine Fault Tolerant (BFT) consensus algorithm to compute A. Classical BFT protocols, however, aim to ensure that at the end of a protocol invocation, all honest nodes store the same value, e.g., in a blockchain, that all nodes store the same fresh block. In our oracle setting, the goal is slightly different. We want to ensure that TRIATICS-SC (and then USER-SC) obtains aggregate answer A = Agg(A1, A2, . . . , An) without participating in the consensus protocol and without needing to receive answers from multiple oracles.

The problem of freeloading, still needs to be addressed.

The Triatics system proposes the use of a simple protocol involving threshold signatures. Such signatures can be realized using any of a number of signature schemes, but are especially simple to implement using Schnorr signatures [4]. In this approach, oracles have a collective public key pk and a corresponding private key sk that is shared among O1,O 2, . . . ,O n in a (t, n)-threshold manner [3]. Such a sharing means that every node Oi has a distinct private / public keypair (ski, pki). Oi can generate a partial signature σi = Sigski [Ai] that can be verified with respect to pki.

The key feature of this setup is that partial signatures on the same value A can be aggregated across any set of t oracles to yield a single valid collective signature Σ = Sigsk[A] on an answer A. No set of t 1 oracles, however, can produce a valid signature on any value. The single signature Σ thus implicitly embodies the partial signatures of at least t oracles.

Threshold signatures can be realized naïvely by letting Σ consist explicitly of a set of t valid, independent signatures from individual nodes. Threshold signatures have similar security properties to this naïve approach. But they

provide a significant on- chain performance improvement: They reduce the size and cost of verifying Σ by a factor of t.

With this setup, it would seem that oracles can just generate and broadcast partial signatures until t such partial signatures enable the computation of Σ. Again, though, the problem of freeloading arises. We must therefore ensure that oracles genuinely obtain data from their designated sources, rather than cheating and copying Ai from another oracle. Our solution involves a financial mechanism: An entity PROVIDER (realizable as a smart contract) rewards only oracles that have sourced original data for their partial signatures.

In a distributed setting, determining which oracles qualify for payment turns out to be tricky. Oracles may intercommunicate off-chain and we no longer have a single authoritative entity (TRIATICS-SC) receiving responses and are therefore unable to identify eligible payees directly among participating oracles. Consequently, PROVIDER must obtain evidence of misbehavior from the oracles themselves, some of which may be untrustworthy. We propose the use of consensus-like mechanisms in our solution for Triatics to ensure that PROVIDER does not pay freeloading oracles.

The off-chain aggregation system we propose for Triatics, with accompanying security proof sketches, may be found in Appendix A. It makes use of a distributed protocol based on threshold signatures that provides resistance to freeloading by $f < n/3$ oracles. We believe resistance to freeloading is an interesting new technical problem.

5 Triatics Security Services

Thanks to the protocols we have just described in the previous section, Triatics proposes to ensure availability and correctness in the face of up to f faulty oracles. Additionally, trusted hardware, as discussed in Section 6, is being actively considered as a secure approach toward protecting against corrupted oracles providing incorrect responses. Trusted hardware, however, may not provide definitive protection for three reasons.

First, it will not be deployed in initial versions of the Triatics network. Second, some users may not trust trusted hardware (see Appendix B for a discussion). Finally, trusted hardware cannot protect against node downtime, only against node misbehavior. Users will therefore wish to to ensure that they can choose the most reliable oracles and minimize the probability of USER-SC relying on > f faulty oracles.

To this end, we propose the use of four key security services: a Validation System, a Reputation System, a Certification Service, and a Contract-Upgrade Service. All of these services may initially be run by one company or group interested in launching the Triatics network, but are designed to operate strictly accordingly to Triatics' philosophy of decentralized design. Triatics' proposed security services cannot block oracle node participation or alter oracle responses. The first three services only provide ratings or guidance to users, while the Contract-Upgrade Service is entirely optional for users. Additionally, these services are designed to support independent providers, whose participation should be encouraged so that users will eventually have multiple security services among which to choose.

5.1 Validation System

The Triatics Validation System monitors on-chain oracle behavior, providing an objective performance metric that can guide user selection of oracles. It will seek to monitor oracles for:

- Availability: The Validation System should record failures by an oracle to respond in a timely way to queries. It will compile ongoing uptime statistics.

- Correctness: The Validation System should record apparent erroneous responses by an oracle as measured by deviations from responses provided by peers.

In our initial, on-chain aggregation system in Triatics, such monitoring is straightforward, as all oracle activity is visible to TRIATICS-SC.

Oracles digitally sign their responses, and thus, as a side effect, generate non-repudiable evidence of their answers. Our proposed approach will therefore be to realize the validation service as a smart contract that would reward oracles for submitting evidence of deviating responses. In other words, oracles would be incentivized to report apparently erroneous behavior.

Availability is somewhat trickier to monitor, as oracles of course don't sign their failures to respond. Instead, a proposed protocol enhancement would require oracles to digitally sign attestations to the set of responses they have received from other oracles. The validation contract would then accept (and again reward) submission of sets of attestations that demonstrate consistent non-responsiveness by an underperforming oracle to its peers.
In both the on-chain and off-chain cases, availability and correctness statistics for oracles will be visible on-chain. Users / developers will thus be able to view them in real time through an appropriate front end, such as a dAPP in Ethereum or an equivalent application for a permissioned blockchain.

5.2 Reputation System

The Reputation System proposed for Triatics would record and publish user ratings of oracle providers and nodes, offering a means for users to evaluate oracle performance holistically. Validator System reports are likely to be a major factor in determining oracle reputations and placing these reputations on a firm footing of trust. Factors beyond on-chain history, though, can provide essential information about oracle node security profiles.

These may include users' familiarity with oracles' brands, operating entities, and architectures. We envision the Triatics Reputation System to include a basic on-chain component where users' ratings would be available for other smart contracts to reference. Additionally, reputation metrics should be

easily accessible off-chain where larger amounts of data can be efficiently processed and more flexibly weighted.

For a given oracle operator, the Reputation System is initially proposed as sup- porting the following metrics, both at the granularity of specific assignment types (see Section 2), and also in general for all types supported by a node:

- Total number of assigned requests: The total number of past requests that an oracle has agreed to, both fulfilled and unfulfilled.
- Total number of completed requests: The total number of past requests that an oracle has fulfilled. This can be averaged over number of requests assigned to calculate completion rate.
- Total number of accepted requests: The total number of requests that have been deemed acceptable by calculating contracts when compared with peer responses. This can be averaged over total assigned or total completed requests to get insight into accuracy rates.

High-reputation services are strongly incentivized in any market to behave correctly and ensure high availability and performance. Negative user feedback will pose a significant risk to brand value, as do the penalties associated with misbehavior. Consequently, we anticipate a virtuous circle in which well-functioning oracles develop good reputations and good reputations give rise to incentives for continued high performance.

5.3 Certification Service

While our Validation and Reputation Systems are intended to address a broad range of faulty behaviors by oracles and is proposed as a way to ensure system integrity in the vast majority of cases, Triatics may also include an additional mechanism called a Certification Service. Its goal is to prevent and/or remediate rare but catastrophic events, specifically en bloc cheating in the form of Sybil and mirroring attacks.

Sybil and mirroring attacks

Both our simple and in-contract aggregation protocols seek to prevent freeloading in the sense of dishonest nodes copying honest nodes' answers. But neither protects against Sybil attacks [9]. Such attacks involve an adversary that controls multiple, ostensibly independent oracles. This adversary can attempt to dominate the oracle pool, causing more than f oracles to participate in the aggregation protocol and provide false data at strategic times, e.g., in order to influence large transactions in high-value contracts. Quorums of cheating oracles can also arise not just under the control of a single adversary, but also through collusion among multiple adversaries. Attacks or faults involving > f oracles are especially pernicious in that they are undetectable from on-chain behavior alone.

Additionally, to reduce operational costs, a Sybil attacker can adopt a behavior called mirroring, in which it causes oracles to send individual responses based on data obtained from a single data-source query. In other words, misbehaving oracles may share data off-chain but pretend to source data independently. Mirroring benefits an adversary whether or not it chooses to send false data. It poses a much less serious security threat than data falsification, but does slightly degrade security in that it eliminates the error correction resulting from diversified queries against a given source Src. Sybil attacks resulting in false data, mirroring, and collusion in general may be eliminated by the use of trusted hardware in our long-term strategy (see Section 6).

Certification Service design

The Triatics Certification Service would seek to provide general integrity and availability assurance, detecting and helping prevent mirroring and colluding oracle quorums in the short-to-medium term. The Certification Service would issue endorsements of high-quality oracle providers. We emphasize again, as noted above, that the service will only rate providers for the benefit of users. It is not meant to dictate oracle node participation or non-participation in the system. The Certification Service supports endorsements based on several features of oracle deployment and behavior. It would monitor the Validation System statistics on oracles and perform post-hoc spot-checking of on-chain answers—particularly for high-value

transactions—comparing them with answers obtained directly from reputable data sources.

With sufficient demand for an oracle provider's data, we expect there to be enough economic incentive to justify off-chain audits of oracle providers, confirming compliance with relevant security standards, such as relevant controls in the Cloud Security Alliance (CSA) Cloud Controls Matrix [26], as well as providing useful security information that they conduct proper audits of oracles' source and bytecode for their smart contracts.

In addition to the reputation metrics, automated on-chain and automated off- chain systems for fraud detection, the Certification Service is planned as a means to identify Sybil attacks and other malfeasance that automated on-chain systems cannot.

5.4 Contract-Upgrade Service

As recent smart contract hacks have shown, coding bulletproof smart contracts is an extremely challenging exercise [1], [20], [22]. And even if a smart contract has been correctly programmed, environmental changes or bugs can still result in vulnerabilities, e.g., [2].
For this reason, we propose a Contract-Upgrade Service. We emphasize that use of this service is entirely optional and in control of users.

In the short term, if vulnerabilities are discovered, the Contract-Upgrade Service would simply make a new set of supporting oracle contracts available in Triatics. Newly created requesting smart contracts will then be able to migrate to the new set of oracle contracts.

Unfortunately, existing ones would be stuck with the old, potentially vulnerable set. In the longer term, therefore, TRIATICS-SC would support a flag (MIGFLAG) in oracle calls from requesting contracts indicating whether or not a call should be forwarded to a new TRIATICS-SC should one become available. Set by default (i.e., if the flag is missing) to false, MIGFLAG would enable requesting contracts to benefit from automatic forwarding and thus migration to the new version of TRIATICS-SC. In order to activate forwarding, a user will cause her requesting contract to issue Triatics requests with

MIGFLAG = true. (Users can engineer their smart contracts so that they change this flag upon receiving an instruction to do so
on-chain from an authorized contract administrator.)

Migration of users to new oracle contracts functions as a kind of "escape hatch," something long advocated for by blockchain researchers (see, e.g., [23]) as a mechanism to fix bugs and remediate hacks without resorting to such cumbersome approaches as whitehat hacking [1] or hard forks. Migration to the updated contracts will be visible on the blockchain, and available to audit for users to review before upgrading.

We recognize nonetheless that some users will not feel comfortable with any one group controlling an escape hatch in the form of migration / forwarding. Forced migration could empower the migrating contract's controller, or a hacker who com- promises relevant credentials, to undertake malicious activity, such as changing oracle responses. It is for this reason that requesting contracts have full control of the for- warding feature and can thus opt out of escape-hatch activation. Additionally, in accordance with Triatics' focus on decentralization, we expect that providers will be able to support multiple versions of TRIATICS-SC developed by the community.

5.5 TRA token usage

The Triatics network utilizes the TRA token to pay Triatics Node operators for the retrieval of data from off-chain data feeds, formatting of data into blockchain readable formats, off-chain computation, and uptime guarantees they provide as operators. In order for a smart contract on networks like Ethereum to use a Triatics node, they will need to pay their chosen Triatics Node Operator using TRA tokens, with prices being set by the node operator based on demand for the off-chain resource their Triatics provides, and the supply of other similar resources. The TRA to- ken is an ERC20 token, with the additional ERC223 "transfer and call" functionality of transfer (address, unit, 256, bytes), allowing tokens to be received and processed by contracts within a single transaction.

6 Long-Term Technical Strategy

The long-term technical strategy for Triatics proposed in this whitepaper includes three key directions: Oracle confidentiality, infrastructure changes, and off-chain computation.

6.1 Confidentiality

A distributed oracle network aims to offer a high degree of protection against faulty oracles. In most deployment scenarios, it seeks to attain a correct response in the face of f Byzantine faults (for $f < n/2$ in our simple aggregation protocol). Trusted hardware can offer much more and is proposed as a better approach to securing the Triatics network. Trusted hardware is the keystone of the Town Crier (TC) oracle [24], which is currently operating on the Ethereum mainnet [33] and whose creators partnered with SmartContract in the TC launch.

Certain forms of trusted hardware, most notably Intel's recent Software Guard eXtensions (SGX) set of instruction-set architecture extensions [12]–[15], [18], seek to provide a powerful adjunct to distributed forms of trust. Briefly, SGX permits an application to be executed in an environment called an enclave that claims two critical security properties.

First, enclaves protect the integrity of the application, meaning its data, code, and control flow, against subversion by other processes. Second, an enclave protects the confidentiality of an application, meaning that its data, code, and execution state are opaque to other processes. SGX seeks to protect enclaved applications even against a malicious operating system, and thus against even the administrator of the host on which an application is running.

While alternative forms of trusted hardware, such as ARM TrustZone, have been in existence for some time, SGX provides an additional key feature lacking in these technologies. It enables a platform to generate an attestation to the execution of a particular application (identified by a build of its hash state). This attestation can be verified remotely and allows a specific application instance to be bound to a public key and thus to establish authenticated and confidential channels with other parties. Running an

oracle in an enclave and distributing attestations can provide very strong assurance that the oracle is executing a particular application, specifically one created or endorsed by developers in the Triatics ecosystem.

Additionally, an oracle running in an enclave that can connect to a data source via HTTPS can provide a strong assurance that the data it retrieves has not been tampered with [24], [33]. These properties go a long way toward protecting against oracle misbehavior in the sense of data corruption, Sybil attacks, etc.

A still greater opportunity, however, lies in the ability of trusted hardware to provide strong confidentiality. The need for confidentiality is in general one of the main hurdles to blockchain deployment. Confidentiality-preserving oracles can be instrumental in solving the problem.

Why distributed oracles don't ensure confidentiality

Confidentiality is fundamentally hard to achieve in any oracle system. If an oracle has a blockchain front end such as a smart contract, then any queries to the oracle will be publicly visible. Queries can be encrypted on-chain and decrypted by the oracle service, but then the oracle service itself will see them. Even heavyweight tools such as secure multiparty computation, which permits computation over encrypted data, can't solve this problem given existing infrastructure [11]. At some point a server needs to send a query to a target data source server. It must see the query, irrespective of whatever confidentiality the query previously enjoyed. It will also see the response to the query.

Confidentiality-preserving oracles via SGX

An oracle using SGX can ingest and process data within an enclave, in essence acting like a TTP trusted for integrity and confidentiality. To begin with, such an oracle can decrypt queries within its enclave. It can then process them without exposing them to any other process (or any human being). The enclave can also process data from sources confidentially and can securely manage sensitive information such as user credentials, a powerful capability, as we illustrate below.

The Town Crier system supports confidential flight data queries. Flight information can be passed to a TC smart contract front end encrypted under the public key of the TC service. TC decrypts the query and then contacts a data source (e.g., flightaware.com) over HTTPS. It returns to the querying smart contract a simple yes/no answer to the question "Has this flight been delayed?" and exposes no other information on-chain.

An even more interesting TC capability is its support for trading on the Steam gaming platform. TC can securely ingest user credentials (passwords) to check that game ownership has been transferred from a buyer to a seller. It can thereby create a secure marketplace that would be otherwise unachievable, with high assurance fair swaps of cryptocurrency for digital goods. (A simple distributed oracle, in contrast, could not securely manage users' passwords on their behalf.)

TC can also perform trusted off-chain aggregation of data from multiple sources, as well as trusted computation over data from multiple sources (e.g., averaging) and interactive querying of data sources (e.g., searching the database of one source in response to the answer of another).
Trusted hardware offers an exciting new approach to the scalable usage of blockchains [24], [29], in which large portions of blockchain infrastructure, including smart contracts, execute in enclaves. Such an architecture would combine transparency benefits of blockchains with the confidentiality properties of off-chain execution and trusted hardware. While similar ideas have been suggested using other techniques, such as zk-SNARKs [21], trusted hardware is far more practical (and less complicated). Our current research agenda includes this expansive vision, with oracles as a catalyzing service.

Defining security given SGX

It is possible, given the use of trusted hardware, to define oracle correctness more formally, starting with the formalism for Intel SGX proposed in [32]. This formalism allows SGX to be treated as a global Universally Composable (UC) [6] functionality sgx(Σsgx)[progencl,].

Here, and in what follows, Σ denotes a signature scheme with signing and verification functions Σ.Sign and Σ.Verify. An instance of Fsgx(Σsgx)[progencl, R] is parameterized by a group signature scheme Σsgx. Argument progencl denotes the program running in an enclave, i.e., environment protected by the hardware. denotes the untrusted code running on an SGX host, i.e., the software that calls the application running in the enclave.

Below (taken from [24]) shows the operation of the functionality Fsgx. Upon initialization, it runs outp := progencl.Initialize(), generating and attestation to the code of progencl and outp. An attestation σatt is a digitally signed statement by the platform that progencl is running in an enclave and has yielded output outp. In typical usage, progencl.Initalize() outputs an instance-specific public key that can be used to create a secure channel to the application instance. Upon a resume call with (id, params), Fsgx continues execution and outputs the result of progencl.Resume(id, params), where id denotes a session identifier and params denotes parameters input to progencl.

$F_{\mathbf{sgx}}[\text{prog}_{\mathbf{encl}}, R]$**: abstraction forSGX**

Hardcoded: sk_{sgx} (private key for Σ_{sgx})

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

Initialize:

On receive (init) from *R*:
  Let outp := $\text{prog}_{\text{encl}}$.**Initalize**()
  σ_{att} := Σ_{sgx}.Sign(sk_{sgx}, ($\text{prog}_{\text{encl}}$, outp))

Resume:

On receive (resume, id, params) from *R*:
  Let outp := $\text{prog}_{\text{encl}}$.**Resume**(id, params)
  Output outp

6.2 Infrastructure changes

Many of the challenges in constructing secure oracles arise from the fact that existing data sources don't digitally sign the data they serve. If they did, then oracles would not need to be trusted to refrain from tampering with data. HTTPS, the protocol for secure web communications, does not enable data signing. It does, though, have an underlying public-key infrastructure (PKI) that requires servers to possess certificates that could in principle support data signing.

This observation is the basis of TLS-N, a TLS extension, that allows HTTPS servers to sign portions of their sessions with clients. The selective nature of the signing provides other nice features, such as the ability for clients to exclude from signed transcripts and thus protect the confidentiality of credentials (e.g., passwords) they use to connect to servers.

We believe infrastructure changes such as TLS-N are promising approaches to supporting oracle security. They will probably need to be used in concert with other technologies such as SGX, however, because of the following limitations:

Infrastructure modifications: Unfortunately, until and unless TLS-N becomes a standard, data sources must expressly deploy it for clients to benefit. Few data sources are likely to in the near future.

Aggregation and computation: TLS-N cannot support aggregation or other forms of trusted computation over data from data sources, so some trusted mechanism will still be required to accomplish these tasks.

Cost: Verification of TLS-N-signed data incurs relatively high on-chain costs compared with simple signature verification.

Confidentiality: TLS-N cannot support out-of-band confidential management of user credentials or queries, but instead requires users to query a data source themselves for this purpose. For example, confidential flight information cannot be stored in a smart contract for later confidential automated query of a website.

6.3 Off-chain computation

Some intriguing uses of oracles, such as the use of credential-dependent APIs, require that an oracle do considerably more than just transmit data. It may need to manage credentials, log into accounts to scrape data, and so forth. Indeed, given truly trust- worthy and confidential oracles, something that SGX-backed systems à la Town Crier and techniques such as zero-knowledge proofs [21] can help achieve, the boundary between oracles and smart contracts may become fluid.

Triatics already supports a regex-based language for queries that enables users to flexibly specify the processing of off-chain data. Our long-term strategy, however, seeks to create a world where oracles are a key off-chain computation resource used by most smart contracts. We believe this will be enabled by building towards a model of fully general, private off-chain computation within oracles whose results are consumed by smart contracts. If this can be achieved with strong security, as we believe it can, pushing costly and sensitive computation logic into oracles will result in better confidentiality, lower contract execution costs, and more flexible architectures.

7 Existing Oracle Solutions

Triatics is designed to fill a pervasive need for new oracle technology in smart contract systems. Today there is a very limited supply of highly secure and flexible oracle systems. We believe this lack of trustworthy oracles is a major impediment to the evolution of smart contracts.

The most commonly used option for oracle services today are centralized oracle providers. This approach is problematic as it creates a centralized point of control, and thus does not meet the high standards of tamper resistance that trustless smart contracts require. Some such systems, e.g. [31], attempt to remedy this problem by relying on notarization, to "prove" correct behavior. This use of notarization services is worrisome in view of documented problems with these services [37], and the fact that their attestations cannot be feasibly verified on-chain, resulting in a (potentially recursive) need for further verification.

Another approach to delivering trustworthy oracle data is to rely on manual human input of unstructured data. These "manual-input oracle" are commonly proposed for use in prediction-markets [17], [25], [28]. By creating appropriate financial stakes and assuming economically rational players with limited financial incentives for cheating, such oracles provide a high assurance of correct crowd-sourced answers. This approach is decentralized and flexible. Since manual-input oracles obtain their responses from human beings, they can respond to questions for which structured data is hard to find, or difficult to extract in a reliable way, e.g., requires natural language processing of news events.

Unfortunately, though, because human cognition is costly and slow, manual-input oracles are resource-intensive, not real-time, and can handle only a limited set of questions at any given time. We believe that Triatics could also be very useful for quickly and automatically resolving prediction-market contracts that can be resolved by structured data.

A final approach is to change the form of data at the source. If a data source digitally signed the data it provided, then the relaying server wouldn't need

to be trusted. USER-SC could simply check the signatures on data it receives. An excellent, general approach of this kind is provided by TLS-N, as discussed above.

8 Conclusion

We have introduced Triatics, a decentralized oracle network for smart contracts to securely interact with resources external to the blockchain. We have outlined the Triatics architecture, describing both on and off-chain components. After defining security in the context of oracles, we described Triatics's multilayered approach to decentralization. We proposed a novel protocol with new features such as protection against freeloading (with additional protocols and security-proof sketches in the paper appendix). We also laid out a roadmap for how Triatics can harness technological and infrastructural advances, such as trusted hardware and digital signing of data by sources. Finally, having examined existing oracle solutions and their shortcomings, we have exposed the need today for a system such as Triatics.

Design Principles. As we continue our work on Triatics, we will seek to prioritize the following core values:

Decentralization for secure and open systems.
Decentralization is not only the foundation of the tamperproof properties of blockchains, but the basis of their permissionless nature. By continuing to build decentralized systems, we aim to further enable permissionless development within the ecosystem. We believe that decentralization is a crucial component for a globally thriving ecosystem with long-term sustainability.

Modularity for simple, flexible system design.
We appreciate the philosophy of building small tools which do one thing well. Simple components can be easily reasoned about and thus securely combined into larger systems. We believe that modularity not only enables upgradable systems, but facilitates decentralization. Wherever key pieces of Triatics depend on or are managed by too few parties, we will seek to design an ecosystem which allows for competing implementations to be used.

Open source for secure, extensible systems.
Triatics is made possible by standing on the shoulders of many open source projects. We value the community and will continue to contribute by

developing Triatics in an open source manner. We plan to engage continually with developers, academics, and security experts for peer review. We encourage testing, audits, and formal proofs of security, all with the aim of creating a platform whose robustness and security can support future innovations.

References

[1] Parity. The Multi-sig hack: A postmortem. https://blog.ethcore.io/the-multi- sig-hack-a-postmortem/. 20 July 2017.
[2] Gun Sirer. Cross-Chain Replay Attacks. Hacking, Distributed blog. 17 July 2016. [3]Adi Shamir. “How to share a secret”. In: Communications of the ACM 22.11
(1979), pp. 612–613.
[4] Claus-Peter Schnorr. “Efficient signature generation by smart cards”. In: Journal of cryptology 4.3 (1991), pp. 161–174.
[5] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, et al. “Secure distributed key generation for discrete-log based cryptosystems”. In: Eurocrypt. Vol. 99. Springer. 1999, pp. 295–310.
[6]R. Canetti. “Universally Composable Security: A New Paradigm for Crypto- graphic Protocols”. In: FOCS. 2001.
[7] Ran Canetti. “Universally composable security: A new paradigm for crypto- graphic protocols”. In: Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on. IEEE. 2001, pp. 136–145.
[8] Douglas R Stinson and Reto Strobl. “Provably secure distributed Schnorr signa- tures and a (t, n) threshold scheme for implicit certificates”. In: ACISP. Vol. 1. Springer. 2001, pp. 417–434.
[9] John R Douceur. “The sybil attack”. In: International Workshop on Peer-to- Peer Systems. Springer. 2002, pp. 251–260.
[10] Aniket Kate and Ian Goldberg. “Distributed key generation for the internet”.
In: Distributed Computing Systems, 2009. ICDCS’09. 29th IEEE International Conference on. IEEE. 2009, pp. 119–128.
[11] Claudio Orlandi. “Is multiparty computation any good in practice?” In: Acous- tics, Speech and Signal Processing (ICASSP), 2011 IEEE International Con- ference on. IEEE. 2011, pp. 5848–5851.
[12] Ittai Anati, Shay Gueron, Simon Johnson, et al. “Innovative technology for CPU based attestation and sealing”. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. Vol. 13. 2013. URL: https : / / software . intel . com / en - us / articles / innovative- technology- for- cpu- based- attestation- and- sealing (vis- ited on 05/23/2016).

[13] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, et al. “Using Innovative Instructions to Create Trustworthy Software Solutions”. In: Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Se- curity and Privacy. HASP ’13. Tel-Aviv, Israel: ACM, 2013, 11:1–11:1. ISBN: 978-1-4503-2118-1. DOI: 10.1145/2487726.2488370. URL: http://doi.acm. org/10.1145/2487726.2488370.
[14] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, et al. “Innovative instruc- tions and software model for isolated execution.” In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. 2013, p. 10. URL: http://css.csail.mit.edu/6.858/2015/ readings/intel-sgx.pdf (visited on 05/23/2016).
[15] Intel. Intel Software Guard Extensions Programming Reference. 2014. (Visited on 05/23/2016).
[16] Gavin Wood. “Ethereum: A secure decentralised generalised transaction ledger”.
In: Ethereum Project Yellow Paper (2014).
[17] Jack Peterson and Joseph Krug. “Augur: a decentralized, open-source platform for prediction markets”. In: arXiv preprint arXiv:1501.01042 (2015).
[18] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: Cryptology ePrint Archive (2016). URL: https : / / eprint . iacr . org / 2016 / 086 . pdf (visited on 05/24/2016).
[19] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. “Sanctum: Minimal Hard- ware Extensions for Strong Software Isolation.” In: USENIX Security Sympo- sium. 2016, pp. 857–874.
[20] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, et al. “Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab”. In: International Conference on Financial Cryptography and Data Security. Springer. 2016, pp. 79–94.
[21] Ahmed Kosba, Andrew Miller, Elaine Shi, et al. “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts”. In: S&P’16. IEEE. 2016.
[22] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, et al. “Making smart contracts smarter”.

In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM. 2016, pp. 254–269.
[23] Bill Marino and Ari Juels. “Setting standards for altering and undoing smart contracts”. In: International Symposium on Rules and Rule Markup Languages for the Semantic Web. Springer. 2016, pp. 151–166.
[24] Fan Zhang, Ethan Cecchetti, Kyle Croman, et al. “Town Crier: An authenti- cated data feed for smart contracts”. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM. 2016, pp. 270– 282.
[25] Augur project page. https://augur.net. 2017.
[26] CSA Cloud Controls Matrix. URL: https://cloudsecurityalliance.org/group/cloud- controls-matrix. 2017.
[27] Mark Flood and Oliver Goodenough. Contract as Automaton: The Computa- tional Representation of Financial Agreements. https://www.financialresearch. gov/ working- papers/ files/ OFRwp- 2015 - 04 _ Contract- as- Automaton- The- Computational- Representation- of- Financial- Agreements.pdf. Office of Financial Research, 2017.
[28] Gnosis project page. https://gnosis.pm. 2017.
[29] Hyperledger Sawtooth. https://intelledger.github.io/introduction.html. 2017.
[30] Abhiram Kothapalli, Andrew Miller, and Nikita Borisov. “SmartCast: An Incentive Compatible Consensus Protocol Using Smart Contracts”. In: Financial Cryptography and Data Security (FC). 2017.
[31] Oraclize project page. http://www.oraclize.it. 2017.
[32] Rafael Pass, Elaine Shi, and Florian Tramer. “Formal abstractions for attested execution secure processors”. In: Eurocrypt. Springer. 2017, pp. 260–289.
[33] Town Crier Ethereum service. http://www.town-crier.org/. 2017.
[34] Florian Tramer, Fan Zhang, Huang Lin, et al. “Sealed-glass proofs: Using trans- parent enclaves to prove and sell knowledge”. In: Security and Privacy (Eu- roS&P), 2017 IEEE European Symposium on. IEEE. 2017, pp. 19–34.
[35] Vitalik Buterin et al. Ethereum white paper. https://github.com/ethereum/ wiki/wiki/White-Paper.
[36] JSON Schema. http://json-schema.org/.

[37] Hubert Ritzdorf, Karl Wüst, Arthur Gervais, et al. TLS-N: Non-repudiation over TLS Enabling Ubiquitous Content Signing for Disintermediation. IACR ePrint report 2017/578. URL: https://eprint.iacr.org/2017/578.