Airtnt: Fair Exchange Payment for Outsourced Secure Enclave Computations

Mustafa Al-Bassam
University College London
m.albassam@cs.ucl.ac.uk

Alberto Sonnino
University College London
alberto.sonnino@ucl.ac.uk

Michał Król
University College London
m.krol@ucl.ac.uk

Ioannis Psaras
University College London
i.psaras@ucl.ac.uk

ABSTRACT

We present Airtnt, a novel scheme that enables users with CPUs that support Trusted Execution Environments (TEEs) and remote attestation to rent out computing time on secure enclaves to untrusted users. Airtnt makes use of the attestation capabilities of TEEs and smart contracts on distributed ledgers to guarantee the fair exchange of the payment and the result of an execution. Airtnt makes use of off-chain payment channels to allow requesters to pay executing nodes for intermediate "snapshots" of the state of an execution. Effectively, this step-by-step "compute-payment" cycle realises untrusted pay-as-you-go micropayments for computation. Neither the requester nor the executing node can walk away and incur monetary loss to the other party. This also allows requesters to continue executions on other executing nodes if the original executing node becomes unavailable or goes offline.

ACM Reference Format:
Mustafa Al-Bassam, Alberto Sonnino, Michał Król, and Ioannis Psaras. 2018. Airtnt: Fair Exchange Payment for Outsourced Secure Enclave Computations. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The cloud computing model developed during the last two decades and realised by cloud computing providers was built on the premise of compute centralisation. That is, computing power is geographically and administratively concentrated in compute infrastructures of industrial scale, generally called datacenters. The centralisation of computing comes with several drawbacks. Firstly, in a world with an increasing number of devices that have low compute and storage resources, or produce enormous amounts of data [43] at the edge of the network (*e.g.,* IoT devices, autonomous cars, or Chromebooks), computation inevitably needs to be outsourced to external computation nodes. This increased amount of data that needs to be transferred from the edge of the network towards the core (to remote servers in order to be processed) is expected to put significant strain on ISPs, inter-ISP business relationships and the Internet backbone. Recent research has thus been focused on finding ways to bring the servers to the edge, so that most information can stay within the domain boundaries of the edge-ISP rather than having to traverse the Internet backbone [7, 50].

Secondly, cloud computing services act as large central points of failure–for example, Amazon Web Services (AWS) controls up to 40% of the cloud-server market, and when AWS's Virginia datacenter had an outage, a significant part of the web was offline [54]. These central points of control also make it possible for authorities to enforce censorship of specific uses of the cloud, canceling any censorship-resistant property [29].

There is therefore a pressing need to develop alternative, decentralised computing infrastructures. To do so, the features of centralisation need to be re-engineered to fit into a decentralised and distributed computing domain. For instance, in the centralised model of cloud computing, the user and the server provider trust not to defraud each other (*i.e.,* the provider will provide the services that the user paid for, and the user agrees to provide the payment). The user also trusts the server provider to not tamper with the task that the user would like to execute, or its resulting data. The cloud provider achieves that by building reputation arounds its services, while disputes (between users and cloud providers) are resolved through the (physical) court system.

In a decentralised setting, instead, trust and reputation is more difficult to build as any node can join the system, provide services and get paid. That said, users do not know who to trust. Building a reputation system in this case, requires a trusted third party to enforce a "one review per user" rule to prevent Sybils [24]. An important research task is therefore, the following: *"How can we design a decentralised computing platform where executing nodes can execute user's tasks, and receive payment for it ('fair exchange'), without: i) the user and the node trusting each other, ii) a third party to settle disputes and iii) possibility for either party to defraud each other (*e.g., *by lack of payment from the user side or incorrect execution from the compute provider side"*,

Trusted Execution Environments (TEEs) [20, 34, 35], have the potential to address the execution integrity issue, by enabling secure communication between the virtual instance of an application and an external entity, as well as tamper-proof execution (see section 3.3). However by themselves they do not allow for the fair exchange of payment and result for two mutually distrusting parties.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

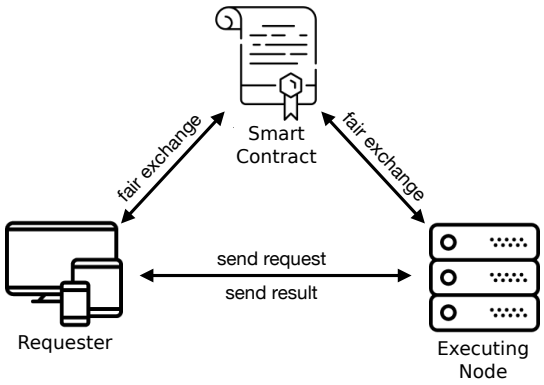

Figure 1: Simplified Airtnt overview.

Fair exchange schemes have traditionally required the participation of a trusted third party [45]. However, trustless distributed ledgers ('blockchains') and smart contracts [15] (see Section 3.1) can take the role of the mediating third party instead, and facilitate the fair exchange of the payment and execution result. By augmenting TEEs so that results attested by the TEE can be fed to a smart contract, fair exchange can be facilitated.

One of the key challenges of this approach however, is to overcome the lack of practicality in sending large results to a smart contract, which all nodes have to verify. Storing large amounts of data in a blockchain is associated with high transaction fees [60]. It would therefore be desirable to have to avoid to store the result at all in the blockchain.

We make the following contributions.

- We design a fair exchange system, called Airtnt, for TEE computation results and payments, according to which a malicious user is not able to defraud others and make them lose money, even if the malicious user is willing to lose money.
- We design all the required protocols and protocol features to support *execution integrity* (*i.e.,* requested computations should be executed correctly without anyone being able to tamper with the execution, and the requester should have a cryptographic proof of the correct execution).
- We design a micropayment system using payment channels for requesters to pay executing nodes as they execute the computation. Requesters receive the changes to the state of the computation after every micropayment and executing nodes receive the micropayment in order to continue to the next state of the computation. This way, the computation can also continue on a different executing node should the original one go offline.
- We implement and evaluate a prototype to demonstrate the practicality of our solution. We implement two separate use-cases, namely, Optical Character Recognition and Game of Life simulations, each of which demonstrates a separate feature of Airtnt.

2 SIMPLIFIED SYSTEM OVERVIEW

In Airtnt, we consider that executing nodes receive requests from requesters to execute predefined functions on some inputs. The mechanism for requesters and executing nodes to discover each other is out of the scope of this paper (*e.g.,* it could work as a web-based service that acts as a marketplace where requesters can bid for executing nodes to execute their requests).

Upon executing a request, an executing node responds to the requester with the result encrypted with a newly generated ephemeral secret key, and the cryptographic hash of the secret key, both generated and attested by the TEE during the execution of the request. TEE attestation ensures that the executing node has computed the correct function and is sending back the correct result. The executing node then submits the secret key itself to a smart contract initialised by the requester. The contract is programmed to send funds to the executing node upon submission of the preimage of the hash of the secret key sent to the requester. This allows the user to decrypt the result, thus achieving the fair exchange of the money and the execution result without trusting a third party (the third party is effectively the smart contract, which facilitates the fair exchange of the payment and execution result, and is untrusted).

Additionally, Airtnt allows for long executions to be 'paused' after certain checkpoints in the execution (which we call 'execution cycles'), returning the intermediate state of the execution of the function as the result to the requester. The requester can then continue the execution by submitting a micropayment through a special type of Airtnt payment channel, off the blockchain (see section 3.2). Micropayments reduce the impact of a requesting node going offline or becoming dishonest after a long execution and wasting the requesting node's resources without payment. It can be thought of as a way to 'livestream computation' with micropayments. The executing node only continues the execution after a micropayment has been received.

The payment channel is closed at the end of the execution, or if there is a dispute. If the executing node goes offline at any point during the execution, the requester can continue the execution with a different executing node using the intermediate state.

3 BACKGROUND

We present background on smart contracts, payment channels and trusted execution environments.

3.1 Smart Contracts on Blockchains

The concept of a blockchain was first proposed in Bitcoin [42], as a decentralised append-only ledger of financial transactions. The Bitcoin blockchain provides a global ordering on the transactions, in order to prevent funds being spent twice (the 'double-spend' attack). As by now there is extensive literature on this topic, we only mention the properties of blockchains that Airtnt relies on.

Bitcoin transactions have a simple internal scripting language that allow the transaction creator to define, as a script, the recipient of the transaction, such that in order for the recipient to spend the funds in the transaction, they must provide an input that causes the script to return *true*. The most common Bitcoin script defines a hash of the public key for the recipient, and returns true upon an input that provides a valid signature associated with that public

key. The hash of the public key is referred to as an 'address', as senders can use it to send funds to the owner of the key.

Blockchain platforms such as Ethereum [15] have extended Bitcoin's script language to allow users to execute more complex programs on the blockchain, called 'smart contracts'. These are interfaces that users can send funds to, such that the management of those funds are defined by the code of the smart contract. Smart contracts in Ethereum can be written in high-level languages (*i.e.,* Solidity, a JavaScript-like language for Ethereum smart contracts), and are compiled down to Ethereum Virtual Machine (EVM) assembly code. This is then published on the blockchain to create instances of smart contracts, which have their own addresses. Like classes in object oriented programming languages, instances of smart contracts have methods that can be called.

Executing a transaction calling a method in an Ethereum smart contract has a 'gas' cost associated with it; the more assembly opcodes and storage the transaction has, the higher the gas cost. The price of gas varies depending on the load on the network, and is paid for using Ether–Ethereum's built-in currency.

Blockchain platforms such as Bitcoin and Ethereum use proof-of-work as a consensus mechanism to agree on the ordering of blocks (which contain batches of transactions that update the state of the ledger), where nodes called miners create new blocks by repeatedly hashing the block until it is below a target value, which is adjusted by the network such that a block is generated every 10 minutes in the case of Bitcoin, or 30 seconds in the case of Ethereum. In the case of a fork, the chain with the most accumulated proof-of-work is considered the correct chain. The security model for proof-of-work blockchains is that in order for a party to undo a transaction, they must create a fork of the chain with more accumulated proof-of-work that the chain that has the transaction in it. In order to do so, they need to be able to expend more mining power than the rest of the network (the '51% attack'), thus making such attacks economically unpractical.

3.2 Payment Channels

The fact that all nodes must verify every transaction in public blockchains, such as Bitcoin or Ethereum, raises scalability issues and limits their usability in practice [14]. Especially for systems such as Airtnt, where micropayments are needed, the total transaction fees would become prohibitive.

As a result, research in this area has shifted to *off-chain payments* [2] using payment channels [3]. This allows two parties to make payments to each other without recording all of their transactions on the blockchain. This typically works by requiring both parties to create an initial transaction on the blockchain and open the channel by depositing a certain amount of coins. For as long as the channel is open, the two parties can make unlimited payments directly to each other without touching the blockchain, effectively updating each other's balance locally. The channel can then be closed by either party, settling the balance on the blockchain with a final transaction. This can be thought of as a similar process to opening a tab at a bar.

There are many technical proposals for designing payment channel systems on top of the Bitcoin blockchain [3]. These proposals are designed so that neither of the two parties can steal funds from each other, or need to trust each other. In Airtnt we focus on unidirectional payment channels using Ethereum, as they are simple to implement as a smart contract, without changes to the protocol.

Unidirectional payment channels only allow one party (the sender) to make payments to the other party (the recipient). In order to close a channel and settle a balance, both parties must sign a message with the owed amount to the recipient and submit it to the smart contract, which will then send the owed coins to the recipient. The typical workflow of a unidirectional payment channel involves the sender sending a signed message to the recipient increasing the owed amount of coins. The channel is updated and the recipient performs the agreed service. Finally, the recipient closes the channel by signing the last update to the channel, and sending the two signatures to the smart contract, releasing the deposited funds.

To prevent the need for a user to have to open a payment channel and maintain a deposit with everyone that they would like to interact with, it is possible to create *multi-hop payment channels*. This allows a payment to travel across multiple users, updating the balances of multiple payment channels. The Lightning network [1] is one example of this for Bitcoin, and the Raiden network [5] is an example for Ethereum.

3.3 Trusted Execution Environments

Traditionally, one protects the integrity and confidentiality of applications by enforcing the isolation of applications. An operating system can isolate applications using hardware mechanisms like virtual address spaces and privileged instructions. Multiple operating systems running on the same physical host are isolated by the hypervisor using hardware virtualisation extensions provided by the CPU. However, Airtnt assumes that functions are executed on untrusted nodes and that both the hypervisor and the OS can be compromised. At the same time, in Airtnt, a running virtual machine requires a private key or a password to decrypt incoming data. The secret must remain confidential and protected against access from the hosting node.

Trusted Execution Environment (TEE) is an environment where code being executed inside the TEE is trusted in authenticity and integrity and both the code and other assets are protected from external access. Multiple TEEs are currently being developed for mobile devices [27] and the most popular CPU architectures [20, 59]

Intel Secure Guard Extension. Intel SGX [20, 34, 35] is an example of Trusted Execution Environment (TEE) that allows applications or part of applications' code to be executed in a secure container, called *enclave*, protecting the integrity and confidentiality of code and data using hardware mechanisms directly in the CPU. SGX enclaves are protected from other applications, privileged system software, such as the operating system (OS), hypervisor, and BIOS, as well as attackers having physical access to the machine. SGX implements hardware encryption in the CPU so that only the authorised enclave can access its region of memory. To enable an application to use enclaves, the developer must provide a signed shared library that will be executed inside an enclave. The library itself is not encrypted and can be inspected before being started, hence no secret should be stored inside the code. The enclave code cannot directly access OS functions (e.g., networking, I/O); it must invoke these functions

through special entry-points that are under strict control of the enclave application.

Remote Attestation. SGX also provides a remote attestation protocol. Using remote attestation an enclave can obtain a statement from the CPU attesting that it is running a particular enclave code with a given memory footprint. A requester can use this attestation to verify the identity and integrity of a target enclave running on a remote host, be convinced that the attestation has been generated by a legitimate SGX CPU, and securely transfer confidential data using an encrypted connection [31]. Intel SGX provides each enclave with a seal key that can be used to store data on permanent storage and access it again upon subsequent execution. This facilitates the development of applications that can restart an enclave without requiring a new remote attestation. The enclave instead loads its secrets from a configuration file encrypted with the enclave specific seal key and kept in stable storage.

Currently, SGX supports only specially prepared Linux and Windows libraries (.so and .dll files), but multiple works extend it to legacy application and containers[1] [11, 12, 53]. In its current version, SGX is susceptible to cache-timing attacks [30], but an enhancement has already been proposed to eradicate this vulnerability [52]. Moreover, SGX does not introduce significant overhead or increase of execution time [61].

4 THREAT MODEL AND GOALS

The following actors participate in an Airtnt transaction:

- **Requesters.** End-user (possibly constrained IoT) devices submit resource-heavy tasks to executing nodes. End-users need to pay the executing node for the CPU cycles that the latter has spent to complete the requested computation.
- **Executing nodes.** High CPU capacity nodes using their resources to execute requested tasks/functions. Executing nodes are paid for carrying out computations.

We assume that both parties mutually distrust one another. Each party is potentially malicious, *i.e.,* they may attempt to steal funds, avoid making payments, and forge results, if it benefits them. At any given time, each party may drop, send, record, modify, and replay arbitrary messages in the protocol.

We assume that each executing node has a TEE-capable machine. Both the requester and the executing node trust their own environments, the TEE, and the function running in the enclave (*i.e.,* the function has been checked before by the user or the community). The enclave is thus trusted to correctly compute and attest the result and not leak information to the hosting execution platform. The rest of the system, such as the network between the parties and the other party's software stacks (outside the TEE) and hardware are untrusted. During function execution, the executing node may therefore: (i) access or modify any data in its memory or stored on disk; (ii) view or modify its application code; and (iii) control any aspect of its OS and other privileged software.

Airtnt makes use of smart contracts on a blockchain to facilitate payments *without a trusted third party*. We assume that the underlying blockchain where the smart contract runs on is resistant to double-spending attacks, and has liveness - that is, transactions submitted to the blockchain will be eventually processed, within some defined period of time.

Given this threat model Airtnt achieves the following goals:

- **Fair exchange.** A requester will only receive the result of an execution if the executing node gets paid for the execution, and vice versa: an executing node will only get paid for the execution, if the requester receives the result.
- **Executing node counterparty risk resistance.** In the event of a requester going offline or diverging from the protocol, a requester cannot cause an executing node to perform a large amount of work without payment.
- **Execution transferability.** In the event of an executing node going offline or diverging from the protocol, a requester should be able to resume a computation on a different executing node, without losing all of the work that the original executing node has already done on the computation.
- **Execution integrity.** The result of computations returned to requesters must be correct and verifiable that the executing node executed the correct program.

5 SYSTEM DESIGN

We describe the Airtnt execution model within a TEE, and how this model can be used for the fair exchange of i) payments and ii) computation results.

5.1 Preliminaries and Notation

We present the notion used in the rest of the paper, as well as the primitives on which Airtnt relies.

Cryptographic Primitives. Airtnt assumes a cryptographically secure hash function $h(x)$. It also assumes a symmetric-key encryption algorithm and a random key generation function $generateKey()$. We denote a message m encrypted with the symmetric key k as $\{m\}_k$.

Smart Contract Primitives. For parts of the scheme that rely on the public-key cryptography native to smart contracts, we denote a message m encrypted with a private key sk as $[m]_{sk}$. We assume a smart contract environment with support for the following primitives:

- **checkSig**$(m, \sigma, addr)$**:** returns $true$ if the signature σ is a valid cryptographic signature of the message m by address $addr$.
- **send**$(v, addr)$**:** sends v coins to the address $addr$.
- **destroy**()**:** terminates the contract, so that no party can call any of the contract's functions.

5.2 Execution Model

We present the Airtnt execution model.

Airtnt Functions. Requesters within Airtnt can request executing nodes to execute Airtnt functions, which are programs loaded in a secure enclave and remotely attested by the TEE. These programs are predefined; the mechanism by which a requester can distribute a program to executing nodes is irrespective of the Airtnt protocol and out of the scope of this work.

[1]Note that only the part of the application that is processing sensitive data needs to run in the enclave—not the whole application.

A requester requests for an executing node to compute the result of an Airtnt function p. These functions take as input a state s, and the number of execution cycles c to perform. They output the new state s', a set of messages representing the output of the program $\vec{o}$, and the number of execution cycles actually performed d. If the execution terminates before c cycles are performed, then $\mathsf{d} < \mathsf{c}$.

$$p(\mathsf{s}, \mathsf{c}) \rightarrow \mathsf{s}', \vec{o}, \mathsf{d} \tag{1}$$

In this model, state s_0 of the function, provided in the beginning, also contains the initial input parameters to the function. However, the function may no longer need to work with the initial input parameters in the middle of the execution, thus we do not explicitly pass them to the function.

The demarcation of execution cycles is arbitrarily defined by the implementation of a function p—as it is the function that is responsible for calculating the number of cycles done d—and does not necessarily correspond to execution cycles on the underlying CPU[2]. In that sense, each execution cycle acts as a checkpoint. Execution cycles in Airtnt play a similar role to gas costs in Ethereum–they are a way to determine the cost of an execution. However an execution cycle on a non-terminal state always results in a new state, so unless s is a terminal state, $\mathsf{s} \neq \mathsf{s}'$ if $\mathsf{c} > 0$ in Equation (1).

If execution of p is split up into multiple steps, it should give the same result and use the same number of execution cycles, such that given $p(\mathsf{s}_0, a) \rightarrow \mathsf{s}_1, \vec{o}_1, \mathsf{d}_1$ and $p(\mathsf{s}_1, b) \rightarrow \mathsf{s}_2, \vec{o}_2, \mathsf{d}_2$, then $p(\mathsf{s}_0, a + b) \rightarrow \mathsf{s}_2, \vec{o}_1 \cup \vec{o}_2, \mathsf{d}_1 + \mathsf{d}_2$.

If after execution of p, s' is not a terminal state resulting from p on s, then the requester may feed s' into p again to continue the execution. Note that Airtnt's novel design allows for the computation to be continued on a different executing node (*e.g.,* in case the original one goes offline).

Wrapper Function. We also define a wrapper function w for p that is executed by the TEE upon receiving a request from a requester. The wrapper function forms the basis of Airtnt's fair exchange protocol. The function takes in as input an Airtnt function p, input state s, and number of execution cycles to perform, c. The function outputs the new state s' encrypted with an ephemeral symmetric key k ($\{\mathsf{s}'\}_k$), the output buffer encrypted with k ($\{\vec{o}\}_k$), the number of cycles performed d, and the cryptographic hash of k (k_{hashed}). The value of these outputs are all attested by the TEE.

$$w(p, \mathsf{s}, \mathsf{c}) \rightarrow \{\mathsf{s}'\}_k, \{\vec{o}\}_k, \mathsf{d}, \mathsf{k}_{hashed} \tag{2}$$

The wrapper function w is the same for any function p, thus a developer writing an application for Airtnt is only concerned with defining p as their program.

Algorithm 1 Do procedure $w(p, \mathsf{s}, \mathsf{c})$

$\mathsf{s}', \vec{o}, \mathsf{d} \leftarrow p(\mathsf{s}, \mathsf{c})$
$\mathsf{k} \leftarrow generateKey()$
$\mathsf{k}_{hashed} = h(\mathsf{k})$
return $\{\mathsf{s}'\}_k, \{\vec{o}\}_k, \mathsf{d}, \mathsf{k}_{hashed}$

[2]It is not always practical to count the number of instructions executed in a program during runtime, and CPU may not be the only resource requirements of a program—some programs may for example be more bandwidth- or memory-intensive.

State Diffs. In many applications, the difference between a given state s and a later state s' may be small, *i.e.,* only a part of the state may change. It would therefore be wasteful, *e.g.,* in terms of bandwidth cost, for the executing node to have to resend parts of an intermediate state that the requester already has knowledge (an identical copy) of. Instead of the executing node sending the whole state to the requester, in Airtnt, the executing node sends a *diff* of the new state to the previous one containing only the changes. We assume a function *genDiff* that can be used by executing nodes to generate such diffs from two states, and a function *applyDiff* that can be used by requesters to decode the new state from a diff. The execution model is independent of the actual implementation of these functions.

$$genDiff(\mathsf{s}, \mathsf{s}') \rightarrow sd \tag{3}$$

$$applyDiff(\mathsf{s}, sd) \rightarrow \mathsf{s}' \tag{4}$$

The wrapper function can then be augmented to return the diff of the new state rather than the entire new state.

Algorithm 2 Do procedure $w(p, \mathsf{s}, \mathsf{c})$

$\mathsf{s}', \vec{o}, \mathsf{d} \leftarrow p(\mathsf{s}, \mathsf{c})$
$\mathsf{k} \leftarrow generateKey()$
$\mathsf{k}_{hashed} = h(\mathsf{k})$
return $\{genDiff(\mathsf{s}, \mathsf{s}')\}_k, \{\vec{o}\}_k, \mathsf{d}, \mathsf{k}_{hashed}$

5.3 Payments Protocol

We describe Airtnt's payments protocol based on the execution model described above. The protocol builds on a smart contract-based unidirectional payment channel between the requester and the executing node.

Firstly, we define a smart contract for creating and managing an Airtnt-specific unidirectional payment channel that enables the requester to send micropayments to the executing node after each intermediate state transition. The contract is held at $id_{contract}$ and is initiated by the function $initChannel(addr_r, addr_e, timeout, deposit)$ containing the following variables:

- $addr_r$: the address of the of requester (corresponding to a public key);
- $addr_e$: the address of the executing node (corresponding to a public key);
- $timeout$: the date that the channel will expire;
- $deposit$: the number of coins that the requester has deposited into the contract.

Additionally, corresponding to addresses $addr_r$ and $addr_e$ are private keys sk_r and sk_e.

After the contract is initialised, the balance (*i.e.,* number of coins owed to the executing node) of the channel can be updated by the requester. To do so, the requester is sending: *i)* a signature to the executing node for the hash of the message $id_{contract}||v||\mathsf{k}_{hashed}$, *ii)* the concatenation of the ID of the contract $id_{contract}$, *iii)* the total amount of coins owed to the requester v, and *iv)* the hash of the secret key k_{hashed} provided by the executing node after the last execution round (see Section 5.2). Note that this procedure takes place off the smart contract platform, as the requester sends

the signature directly to the executing node. We denote the new channel state signed by a private key sk_x as follows:

$$[h(id_{contract}||v||\mathrm{k}_{hashed})]_{sk_x} \quad (5)$$

The executing node can call the smart contract method *closeChannel* by presenting a valid signature for the new state of the channel from the requester, as well as its own signature for the state of the channel (and the new balance of the channel). Crucially, the executing node must provide to the smart contract the pre-image k of k_{hashed} in order to close the channel and receive the funds, so that the requester can decrypt the computation result, as will be discussed in Section 5.4.

Algorithm 3 Do procedure $closeChannel(sig_r, sig_e, v, \mathrm{k}_{hashed}, k)$

$statehash \leftarrow h(id_{contract}||v||h(\mathrm{k}))$
assert $checkSig(statehash, sig_r, addr_r)$
assert $checkSig(statehash, sig_e, addr_e)$
assert $\mathrm{k}_{hashed} = h(k)$
assert $v <= deposit$
$send(v, addr_e)$
$send(deposit - v, addr_r)$
$destroy()$

The payment channel also employs a *timeout*, such that if the timeout is reached and the channel has not been closed, the channel can be destroyed and deposits are returned back to the senders. This is to prevent the case where the recipient of the channel (executing node in our case) is malicious and extorts the sender of the channel (requester in our case) to transfer extra money (without doing any work) or otherwise, risking losing their deposit. The malicious receiver/executing node can achieve that by just refusing to close the channel. If this happens, the requesting node can simply wait until the timeout to recover the deposited funds.

We thus define a smart contract procedure *channelTimeout* that can be called after the timeout is reached.

Algorithm 4 Do procedure $channelTimeout()$

if $timeout > now$ **then**
  $send(deposit, addr_r)$
  $destroy()$
end if

5.4 Airtnt protocol

We describe the overall Airtnt protocol unifying the execution model described in Section 5.2 and the payments protocol described in Section 5.3. Figure 2 provides an an overview of the protocol.

Setup phase. This phase is performed once at the beginning of every relationship between a requester and an executing node for a specific Airtnt function.

① A requester R sends p, s, c (the function to be executed, the initial state, and the number of cycles to perform) to an executing node E. R also specifies some payment rate per cycle, $rate$, that defines the number of coins that R is willing to pay E for each computational cycle performed.

② E responds to R by either accepting or rejecting the request.

③ If E agrees to fulfill the request, then R initializes a payment channel at $id_{contract}$, with the parameters $initChannel(addr_R, addr_E, timeout, deposit)$ where $addr_R$ is the address of the requester, $addr_E$ is the address of the executing node, $timeout$ is some conservative timeout that is longer than the time expected to fulfil the request, and $deposit$ is a value greater or equal to $rate * \mathrm{c}$. If the payment channel is not initialised with the appropriate parameters, E does not participate, and R can proceed to the termination phase. Both parties also locally maintain a variable b which represents the balance of the channel so far, and is iniitalised with $b = 0$.

Execution phase. This is the phase where the main execution loop takes place.

❶ E executes $w(p, \mathrm{s}, \mathrm{c})$ in the TEE, returning $\{\mathrm{s}'\}_k$, $\{\vec{o}\}_k$, d, k_{hashed}, which are all attested by the TEE, and sends R these outputs.

❷ R checks that these outputs are attested by the TEE. If they are, then R signs a new update to the payment channel, $[h(id_{contract}||b + \mathrm{d} * rate||\mathrm{k}_{hashed})]_{sk_R}$, and sends it to E. R then locally updates b to $b + \mathrm{d} * rate$.

❸ Upon E verifying that the signature received from R is valid for the message $h(id_{contract}||b + \mathrm{d} * rate||\mathrm{k}_{hashed})$, E sends R the secret key k. Note that E does not need to receive the message itself, as it already has the information to compute it. E then locally updates b to $b + \mathrm{d} * rate$. R then decrypts $\{\mathrm{s}'\}_k$ and $\{\vec{o}\}_k$ to get s' and $\vec{o}$.

❹ R can now either request E to continue the execution of p on the new state s', or terminate the protocol and find a different executing node to continue the execution on s', if s' is not the terminal state.

Termination phase. There are two ways to terminate the protocol.

⛔ E can close the payment channel by calling $closeChannel(sig_R, sig_E, v, \mathrm{k}_{hashed}, k)$, thus transferring the owed amount v to E and terminating the contract.

⛔ If E becomes unresponsive or goes offline, then R can wait until the timeout, and call $channelTimeout()$ to get its deposit back.

5.5 Evaluation

We evaluate how the design of Airtnt meets the threat model and design goals set out in Section 4.

Fair Exchange. Airtnt aims to facilitate the fair exchange of payments and execution results. We break up this goal into two security properties, that we argue for: i) E will only get paid iff R receives the result of the execution and ii) R will only receive the result of an execution iff E can get paid for it. For the latter goal we say "can" rather than "will", because it is E's responsibility to close the payment channel and collect the payment, and so if E does not do this then they can voluntarily forfeit the payment, technically speaking.

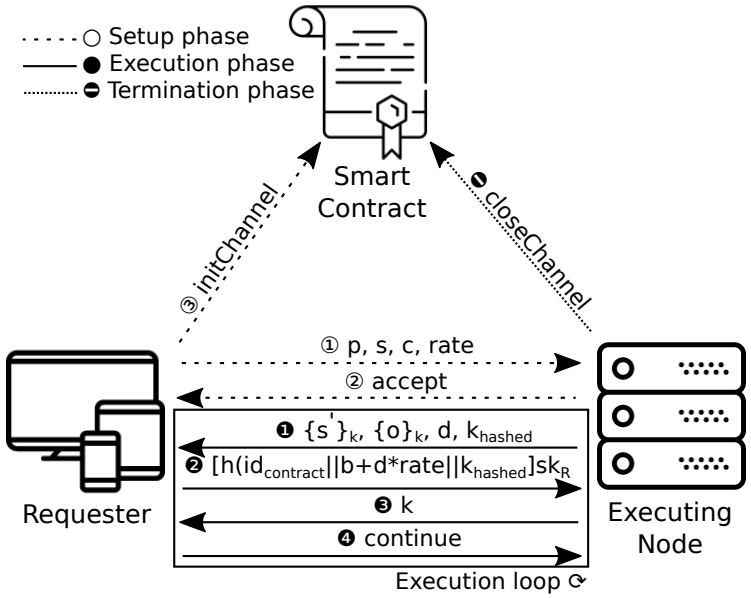

Figure 2: Overview of interactions between a requester, an executing node and a smart contract in Airtnt.

Theorem 5.1. *Assuming R follows the protocol, an executing node E will only get paid for executing* $w(p, \mathrm{s}, \mathrm{c}) \rightarrow \{\mathrm{s}'\}_k, \{\vec{o}\}_k, \mathrm{d}, \mathrm{k}_{hashed}$ *if and only if the requester R receives* $\mathrm{s}', \vec{o}$.

Proof. (Informal.) The only *send* call in the smart contract that sends any coins to $addr_E$ is in the *closeChannel* function. For b coins to be sent to $addr_E$ via *closeChannel*, both the E and R must produce a signature of a message (the state hash) that authorises the transfer of b to $addr_E$. However, assuming R follows the protocol, then R will only produce such a signature if and only if it receives k from E to successfully decrypt $\{\vec{o}\}_k, \mathrm{d}$ with a correct attestation from the TEE. Furthermore, under the threat model described in Section 4, the TEE is trusted to correctly execute w and produce the encryptions of $\{\vec{o}\}_k, \mathrm{d}$; only the TEE is capable of producing such attestation. □

Theorem 5.2. *Assuming E follows the protocol and can submit a closeChannel transaction that is executed before the channel timeout, R can only receive* $\mathrm{s}', \vec{o}$ *for an execution* $w(p, \mathrm{s}, \mathrm{c}) \rightarrow \{\mathrm{s}'\}_k, \{\vec{o}\}_k, \mathrm{d}, \mathrm{k}_{hashed}$ *if and only if E can get paid for the execution.*

Proof. (Informal.) Assuming E follows the protocol, E will only send k, which is necessary to derive $\mathrm{s}', \vec{o}$ from $\{\mathrm{s}'\}_k, \{\vec{o}\}_k$, to R if and only if E receives from R a signed update to the channel for the new balance b that pays for the execution. This signed update to the channel state enables E to send a *closeChannel* transaction to the blockchain to get paid b coins, if it is called before the channel timeout. Under the threat model described in Section 4, the underlying blockchain network is assumed to be resistant to double spend attacks and will process the underlying transaction before the timeout (has liveness). □

Executing Node Counterparty Risk Resistance. We argue that R can only cause E to execute a single execution round of c cycles, without payment. Note that even if this happens, the fair exchange property of the protocol is not violated: R must make a payment to receive the result.

Theorem 5.3. *Assuming E follows the protocol, R cannot require E to perform more than one execution* $w(p, \mathrm{s}, \mathrm{c}) \rightarrow \{\mathrm{s}'\}_k, \{\vec{o}\}_k, \mathrm{d}, \mathrm{k}_{hashed}$ *without updating the payment channel balance.*

Proof. (Informal.) If E executes w and sends to R $\{\mathrm{s}'\}_k, \{\vec{o}\}_k, \mathrm{d}, \mathrm{k}_{hashed}$, but does not receive a signed channel update in response, E will not execute w on behalf of any more requests from R until it receives a valid payment channel update, which may be never if the payment channel expires. □

Execution Transferability. This goal is met in step ❹ of the execution phase of the Airtnt protocol, building on the execution mode described in Section 5.2. Upon decrypting the new intermediate state s' of the execution, R can either request the same execution node to continue the execution on s', or find a different node to request the execution of the function on s'.

Execution Integrity. The integrity of execution results rely on the security properties of the underlying TEE and on the cryptography underlying the remote attestation protocol. The TEE is trusted to correctly execute programs and produce attestations for their results. If, for example, the TEE has security vulnerabilities, the integrity of the results may be violated. Additionally, in the case of Intel SGX, Intel is a trusted third party, as they have the ability to use their cryptographic keys to generate fraudulent remote attestations.

Recall that in step ❷ of the execution phase of the Airtnt protocol, R must check that the outputs sent by E are attested by the TEE, thus

if the security properties of the TEE hold, then execution integrity also holds.

6 IMPLEMENTATION AND PERFORMANCE

Recall in Section 5.2 that executing nodes can return diffs of the new state, thus programs that manipulate state have different network overheads than programs that only consume state. We prototyped Airtnt for those two types of applications which have different properties in terms of the way they handle state:

- **State-based programs (*i.e.,* simulations).** In simulations and other types of programs that rely heavily on the need to remember and manipulate intermediate state, Airtnt executing nodes need to return the changes made to the state to requesters.
- **Pure programs.** These programs do not need to modify state, but only process it and return output based on the state. After processing parts of the state, the program may discard the state if it does not need to revisit it. Intermediate states may be thus a subset of the input state, with only state removed and not added. These diffs would be negligible in size compared to state-based programs, as they only contain instructions to remove state at specific locations, and do not need to transfer new state data. In some cases, such as frame-by-frame image processing, these programs can be parallelized across multiple Airtnt executing nodes.

We implement and evaluate a Game of Life simulation [19] to illustrate an example of a state-based program, and a simple Optical Character Recognition (OCR) [26] to illustrate an example of a pure program. We release both of these applications as open-source projects on GitHub[3]. We instantiated the requester on Amazon AWS [8] on a *t2.xlarge* instance, and the executor on a Dell XPS laptop with Intel Core i7-8550U running Intel SGX [20] as TEE.

6.1 State-based programs

To illustrate state-based programs we implement and evaluate a Game of Life simulation [19]. The Game of Life is a simple example of simulation acting on an arbitrarily large grid of cells, where each cell is either filled in or not. The game starts with some pattern of filled in cells, and evolves to obtain the next state by applying simple transition rules concurrently to each cell of the gird. We implement this example to show how Airtnt can be used to outsource state-based simulations.

Figure 3 shows how the latency perceived by the requester (the time to receive the final state after first submitting the request) varies with the number of cycles per round (in semi-log scale); this graph captures the total execution time. We fix the total number of simulation steps to 1000 cycles ($c = 1000$), and run the simulation for various grid sizes; 10×10, 25×25 , and 50×50 ($\sim 2.5kB$ of data). As expected, the latency is much higher when requesting a few cycles per round—the requester has to transmit intermediate states back and forth every few cycles, which increases latency. When Airtnt operates by steps of 10 cycles (leftmost point of the graph), the requester has to send and receive 100 intermediate states (the total number of cycles divided by the number of cycles per round)

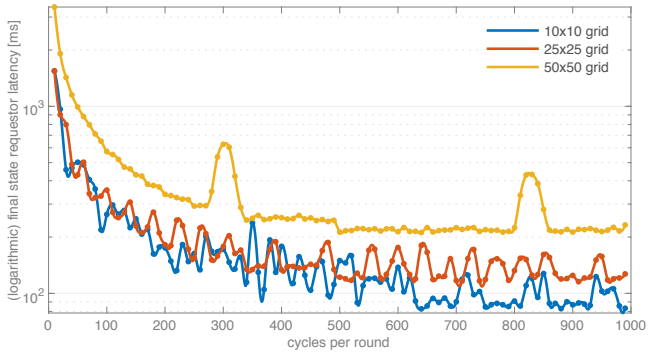

Figure 3: Game of Life – variation of the client perceived latency of receiving the final state with the number of cycles per round for various grid sizes. The total number of cycles is fixed to 1000.

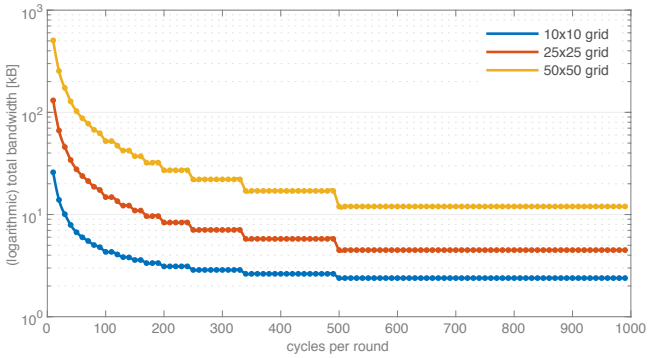

Figure 4: Game of Life – variation of the bandwidth cost with the number of cycles per round for various grid sizes. The total number of cycles is fixed to 1000.

before completing the full simulation of 1000 cycles. However, when operating with number of cycles per round $\in [500, 990]$, only two intermediate states are exchanged, and the latency is about 10 times lower. Moreover, Figure 3 shows that the latency first drops exponentially[4] and then flattens out when increasing the number of cycles per round; increasing it from 10 to 200 reduces the latency by almost 10 times. This allows to achieve latencies of the order of $\sim 100ms$, which is suitable for real-world scenarios. Figure 4 shows the variation of bandwidth cost with the number of cycles (in semi-log scale). As expected, the bandwidth cost decreases with the number of cycles; the number of intermediate states that the requester needs to send varies as the total number of rounds. The bandwidth cost is higher when the number of cycles per round is 10 as the requester sends 100 intermediate states, and then flattens out when the number of cycles per round $\in [500, 990]$ as only 2 intermediate states are transmitted.

Figure 5 shows the enclave execution time per cycle (in semi-log scale). The graph suggests that calling the SGX enclave multiple times introduces overhead. Therefore, working with low values

[3]Removed for submission

[4]Recall that the graph in Figure 3 is in semi-log scale.

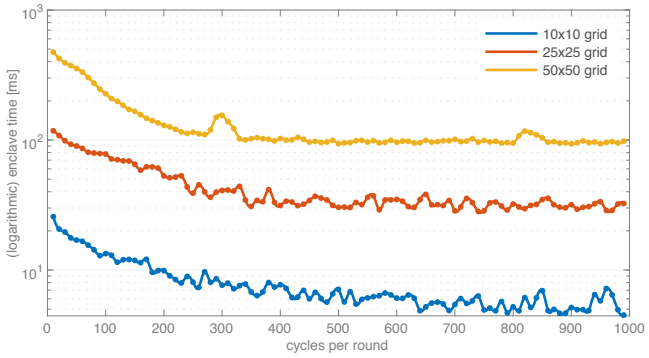

Figure 5: Game of Life – variation of the enclave execution time with the number of cycles per round for various grid sizes. The total number of cycles is fixed to 1000.

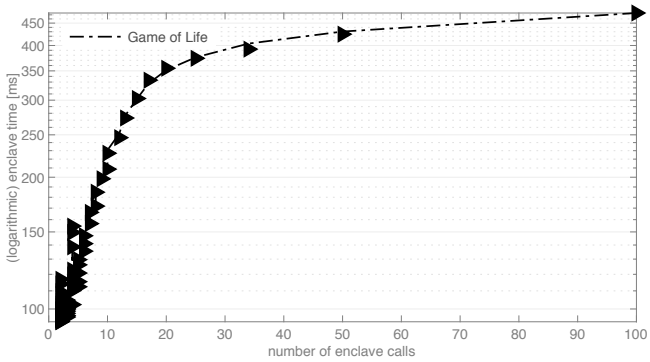

Figure 6: Game of Life – variation of the enclave execution time with the number of enclave calls, for a 50×50 grid. The total number of cycles is fixed to 1000, and the number of enclave calls varies from 2 (for number of cycles per round $\in [500, 990]$) to 100 (for 10 cycles per round).

for the number of cycles per round results in calling the enclave multiple times, which is expensive. To illustrate this phenomena, Figure 6 shows the variation of the enclave execution time with the number of enclave calls. This graph shows that executing 1000 cycles with 2 calls (the number of cycles per round $\in [500, 990]$) is about 5 times faster than executing them with 100 enclave calls (10 cycles per round). Figure 3 and Figure 5 also suggest that when the number of cycles per round is greater than 200 the communication time and the enclave execution time are of the same order of magnitude.

Our evaluation of Game of Life shows a clear trade-off between efficiency and low-risk. Executing Airtnt a few cycles at the time (with low values for the number of cycles per round) helps with mitigating the risk of a requester dropping out without paying for the execution, but at the same time the cost of both bandwidth and execution time increase. However, our evaluations help to find the right balance; as noted above, increasing the number of cycles per round from 10 to 200 improves latency by about 10 times, while increasing it over 300 only adds a small benefit. Therefore, choosing 200 cycles per round seems to be a good choice for our implementation of Game of Life.

6.2 Pure programs

We implement and evaluate a simple Optical Character Recognition (OCR) algorithm [26] to illustrate the execution of pure programs. Our simple algorithm is initialized in the enclave with a model of each letter of the alphabet; it can then be fed with input images to detect the list of embedded letters. We implemented this example to show how Airtnt can be used to outsource execution of pure programs.

Figure 7 shows how the latency perceived by the requester varies with the number of cycles (in semi-log scale). We fix the total number of simulation steps to 1000 cycles in total as in Section 6.1, and run the simulation for images of size $\sim 1kB$. The latency decreases as for the Game of Life simulation but is much higher on overall because the execution of each cycle requires a transmission of a new input image. Therefore, the requester sends about 1.7MB of data to the executor for the processing of 1000 images (1000 cycles)

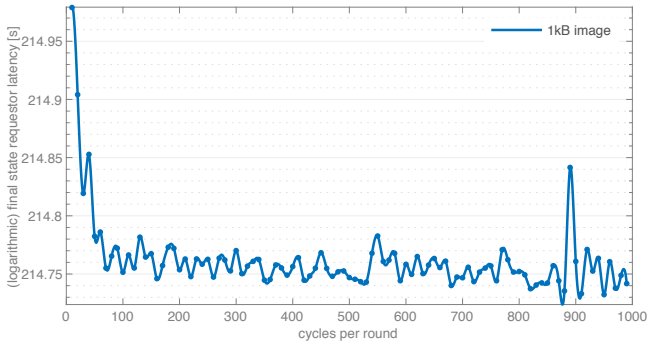

Figure 7: OCR – variation of the client perceived latency on the number of cycles per round for images of $\sim 1kB$. The total number of cycles is fixed to 1000.

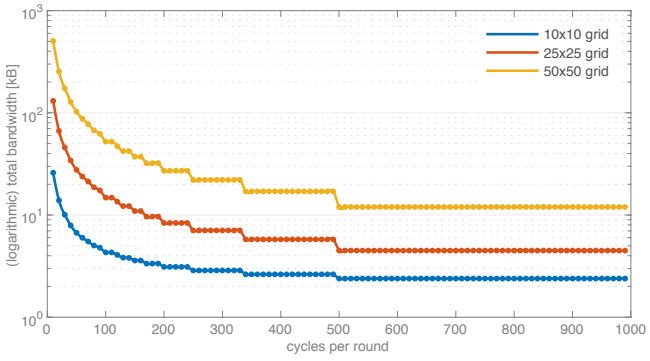

Figure 8: OCR – variation of the bandwidth cost with the number of cycles per round for images of size $\sim 1kB$. The total number of cycles is fixed to 1000.

of 1kB each; and the bandwidth cost stays constant regardless the number of cycles.

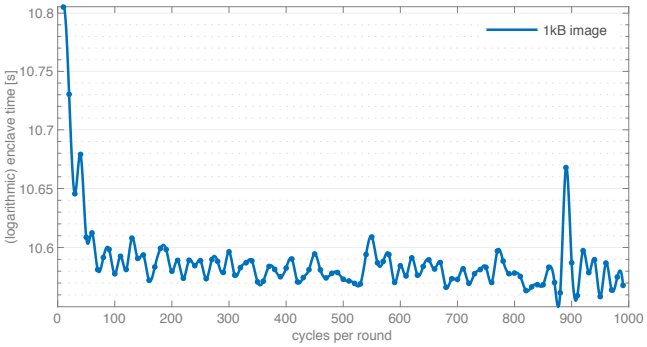

Figure 9: OCR – variation of the enclave execution time with the number of cycles for images of size $\sim 1kB$. The total number of cycles is fixed to 1000.

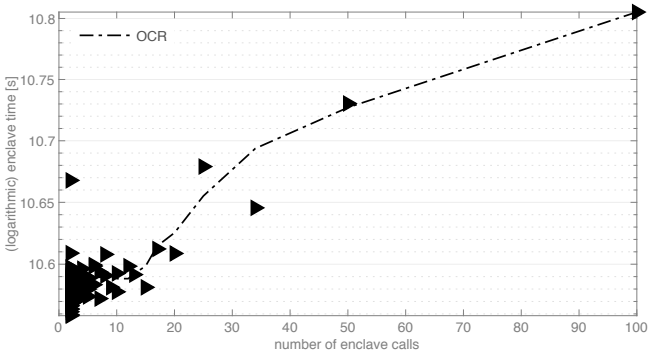

Figure 10: OCR – variation of the enclave execution time with the number of enclave calls, for images of size $\sim 1kB$. The total number of cycles is fixed to 1000, and the number of enclave calls varies from 2 (for number of cycles per round $\in [500, 990]$) to 100 (for 10 cycles per round).

Figure 9 and Figure 10 respectively show the enclave execution time per cycles and per number of enclave calls (in semi-log scale). OCR is a much more CPU-intensive application than Game of Life—running the algorithm over 1000 images requires more than 10 seconds in the enclave. The enclave execution times decreases by 200 ms from 10 cycle per round to 100 cycles per round, and then stays roughly the same. As in Section 6.1, these graphs emphasize the overhead of making multiple calls to the SGX enclave. Moreover, Figure 7 and Figure 9 also suggest that the requester latency is dominated by the time to send the input images (and not by the enclave execution time).

Our experiments illustrate the differences between using Airtnt to outsource the execution of state-based programs (e.g., Game of life) and pure programs (e.g., OCR). State-based programs can run multiple cycles from the same input as they operate on intermediate steps; this allows to save on bandwidth cost as it flattens out when increasing the number of cycles per round. Pure programs require a new input each cycle and cause the bandwidth cost to remain constant; the only benefit of increasing the number of cycles per round in this case is to reduce the number of enclave calls and thus save on execution time. However, pure programs can be parallelized. A requester can send a subset of inputs to many executors, and therefore divide latency by the number of executors involved in the computation.

6.3 Limitations of Intel SGX

In terms of performance, Intel SGX introduces memory limitations and performance overhead. Intel SGX limits the enclave memory to 128 MB; this limitation comes from the BIOS, and sets and upper bound to the input size that the executors can process simultaneously. It is however technically possible to extend that limit by editing the paging support [34]. The performance overhead results from accessing the encrypted memory in an enclave and from the additional effort associated with entering and exiting an enclave. As shown in Section 6, minimizing the number of accesses to the enclave significantly increases performance.

On the security side, relying on Intel SGX requires trust into Intel, as debated by many works recently [21, 36, 37, 49].

6.4 Payment Channel Smart Contract

We implemented an Ethereum smart contract, written in Solidity, for Airtnt unidirectional payment channels.

Due to the Ethereum Virtual Machine having a maximum stack depth of 15, *closeChannel* had to be implemented in such a way that it must be called twice: once for the requester's signature, and once for the executing node's signature. This is because the maximum stack depth limits the number of inputs a function may have. As the maximum size of a variable in the Ethereum Virtual Machine is 32 bytes, the amount of data that a function can take as input is too limited to accept two ECDSA signatures.

We present the gas costs incurred by different functions of the smart contract in Table 1. We note that the price of gas, and the market price of Ether itself, varies wildly from time to time due to volatility, so the USD cost is only accurate as of April 2018. The largest cost is creating the contract at $0.46, as this involves uploading and storing the contract's code in the Ethereum blockchain. However, if the smart contract is uploaded as a library contract, this cost can be significantly reduced as the payment channel code only needs to be uploaded once.

The cost for initialising and closing a channel is $0.10 and $0.15 respectively, thus a complete Airtnt transaction between a requester and executing node would cost $0.25. This cost could be reduced in the future by using multi-hop payment channels (see the Lightning network as an example [47]), so that a requesters do not need to open a payment channel with every executing node, as long as there is a path in the network between the requester and the executing node.

7 RELATED WORK

Result verification has been an active area of research in the past with multiple proposed techniques. The first group of techniques focuses on constructing cryptographic proof of computation [28, 44, 57, 58]. Such proofs are easy to verify without the need to re-execute the computations. However, the overhead of pre-computation and

Method	Gas cost	USD cost
(contract creation)	358,600	$0.46
initChannel	81,053	$0.10
closeChannel	114,757	$0.15
channelTimeout	21,732	$0.03

Table 1: A table showing the gas costs of executing the methods of the payment channel smart contract. The USD costs, which is pegged to the price of Ethereum, are accurate as of April 26, 2018 and assume a gas price of 2 Gwei ($2 \cdot 10^{-9}$ Ether). For *closeChannel*, the figures are for calling *closeChannel* twice, for each respective signature.

creation of the proof is orders of magnitude higher than the actual cost of the computation being verified. The second group of techniques consists of running the same computations on multiple servers [16, 22, 55, 56]. As long as a given fraction of servers is honest, the result can be guaranteed by a consensus protocol. These techniques require at least one honest server and increase significantly the overhead as they rely on repeating the computation.

An approach closer to our work involving blockchain technology is [23]. The authors assume only two execution entities and design smart contracts discouraging them from colluding. Similarly, in [46], authors determine an optimal penalty fee that should be paid by execution platforms caught cheating. Huang *et al.*[33] also distribute the task to multiple workers and exploit Commitment-based sampling [25] to verify the correctness of the result. Before starting the computations, the workers have to commit to the task by spending bitcoins that are lost in case of dishonest behaviour. However, both systems ([23, 33]) still require to repeat computations and assume a trusted 3rd party to resolve conflicts.

Multiple projects focus on incentivising fairness and timely delivery of the results using cryptocurrencies [9, 13, 18, 38, 39]. Workers deposit predefined amount of money that is lost if they misbehave. However, all of them focus on fairness exclusively and ignore verifiability of produced results. Finally, several projects aim to facilitate blockchain-based micro-payments [40, 41, 47]. Those projects are complementary to ours, and could be used to lower the cost and overhead of transactions.

Several systems provide computations verification, but are limited to a specific class of tasks [51]. [17] supports only tasks that can be easily verified by requesters and has lower security (execution platforms are being fairly paid for their job with a probability < 1) and do not provide privacy of the results. Hu *et al.*[32] focus on both task verification and data privacy, but the system does not include payments and works only for computation of the characteristic polynomial of matrix.

Several industrial platforms were launched recently aiming to realise a vision of a global decentralised computer. Golem [48], focuses mainly on video rendering tasks. Execution platforms are automatically paid if the requester confirms completion of the tasks. However, in case of a conflict, the system relies on a consent that must be trusted by both parties. SONM [6] develops a cloud-like services platform based on fog computing as a backend. Execution platforms install SONM OS allowing to share and rent their resources (*i.e.,* CPU, GPU, Storage). So far, SONM do not provide any details about their result verification techniques. iExec [4], yet another platform for result verification and automatic payment announced to use Intel SGX in their system, but does not provide any additional details. Finally, Bounty0x [10] is an open platform allowing people to post tasks with an associated prize. Anyone can then submit a solution and collect the rewards. However, Bounty0x does not provide an automatic verification process and relies on manual conflict resolution.

8 CONCLUSION

We have proposed Airtnt, a protocol that *i)* enables requesters to execute tasks on nodes with TEE-enabled CPUs, and *ii)* allows the executing node to receive payments without either party having to trust each other and without the need for a third party to resolve disputes. We employ smart contracts to act as a trustless mediator for the fair exchange of payment and execution result. We also use checkpoint-based micropayments (through payment channels) to reduce the impact of a requester going offline or becoming dishonest after a long execution and wasting the requesting node's resources without payment.

Our evaluation of Game of Life and OCR show a clear trade-off between efficiency and low-risk. Executing Airtnt a few cycles at a time helps with mitigating the risk of requestors dropping out without paying for the execution, but this comes at the cost of both bandwidth and enclave execution time.

ACKNOWLEDGEMENTS

Mustafa Al-Bassam is supported by a scholarship from The Alan Turing Institute, Alberto Sonnino is supported by the EU H2020 DECODE project under grant agreement number 732546, Michał Król is supported by the EC H2020 ICN2020 project under grant agreement number 723014, and Ioannis Psaras is supported by the EPSRC INSP Early Career Fellowship under grant agreement number EP/M003787/1.

Many thanks to Patrick McCorry for discussions about the Airtnt design.

REFERENCES

[1] 2016. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf. (2016).
[2] 2017. Off-Chain Transactions. https://en.bitcoin.it/wiki/Off-Chain_Transactions. (2017).
[3] 2017. Payment channels. https://en.bitcoin.it/wiki/Payment_channels. (2017).
[4] 2018. iExec Whitepaper. https://iex.ec/whitepaper/iExec-WPv3.0-English.pdf. (2018).
[5] 2018. Raiden Network 101. https://raiden.network/101.html. (2018).
[6] 2018. SONM Documentation. https://docs.sonm.com/. (2018).
[7] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. 2018. Mobile Edge Computing: A Survey. *IEEE Internet of Things Journal* 5, 1 (Feb 2018), 450–465. DOI:http://dx.doi.org/10.1109/JIOT.2017.2750180
[8] Inc." "Amazon Web Services. 2017. AWS Whitepapers. "https://aws.amazon.com/whitepapers/". (2017).
[9] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Secure multiparty computations on bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 443–458.
[10] Adam Angelo, Pascal Thellmann, and Deniz Dalkilic. 2018. Rewarding the Token Economy. https://bounty0x.io/whitepaper_en.pdf. (2018).
[11] Sergei Arnautov and et al. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *OSDI*. 689–703.

[12] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[13] Iddo Bentov and Ranjit Kumaresan. 2014. How to use bitcoin to design fair protocols. In *International Cryptology Conference*. Springer, 421–439.

[14] Ryan Browne. 2017. Big transaction fees are a problem for bitcoin - but there could be a solution. https://www.cnbc.com/2017/12/19/big-transactions-fees-are-a-problem-for-bitcoin.html. (2017).

[15] Vitalik Buterin and others. 2013. Ethereum white paper. (2013).

[16] Ran Canetti, Ben Riva, and Guy N Rothblum. 2011. Practical delegation of computation using multiple servers. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 445–454.

[17] Bogdan Carbunar and Mahesh Tripunitara. 2010. Fair payments for outsourced computations. In *Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference on*. IEEE, 1–9.

[18] Xiaofeng Chen, Jin Li, and Willy Susilo. 2012. Efficient fair conditional payments for outsourcing computations. *IEEE Transactions on Information Forensics and Security* 7, 6 (2012), 1687–1694.

[19] John Conway. 1970. The game of life. *Scientific American* 223, 4 (1970), 4.

[20] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[21] Shaun Davenport. 2014. SGX: the good, the bad and the downright ugly. https://www.virusbulletin.com/virusbulletin/2014/01/sgx-good-bad-and-downright-ugly. (2014).

[22] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. 2016. Resource-efficient Byzantine fault tolerance. *IEEE Trans. Comput.* 65, 9 (2016), 2807–2819.

[23] Changyu Dong, Yilei Wang, Amjad Aldweesh, Patrick McCorry, and Aad van Moorsel. 2017. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. *arXiv preprint arXiv:1708.01171* (2017).

[24] John R Douceur. 2002. The sybil attack. In *International workshop on peer-to-peer systems*. Springer, 251–260.

[25] Wenliang Du, Mummoorthy Murugesan, and Jing Jia. 2010. Uncheatable grid computing. In *Algorithms and theory of computation handbook*. Chapman & Hall/CRC, 30–30.

[26] Line Eikvil. 1993. Optical character recognition. *citeseer. ist. psu. edu/142042. html* (1993).

[27] Jan-Erik Ekberg, Kari Kostiainen, and N Asokan. 2013. Trusted execution environments on mobile devices. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1497–1498.

[28] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. 2016. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1304–1316.

[29] April Glaser. 2018. How Apple and Amazon Are Aiding Chinese Censors. (2018).

[30] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. ACM, New York, NY, USA, Article 2, 6 pages. DOI: http://dx.doi.org/10.1145/3065913.3065915

[31] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive* 2016 (2016), 204.

[32] Xing Hu and Chunming Tang. 2015. Secure outsourced computation of the characteristic polynomial and eigenvalues of matrix. *Journal of Cloud Computing* 4, 1 (2015), 7.

[33] Hui Huang, Xiaofeng Chen, Qianhong Wu, Xinyi Huang, and Jian Shen. 2016. Bitcoin-based fair payments for outsourcing computations of fog devices. *Future Generation Computer Systems* (2016).

[34] Intel. 2014. Software Guard Extensions Programming Reference, Ref. 329298-002US. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf,. (2014).

[35] Intel. 2015. Product Change Notification. https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf. (2015).

[36] Alon Jackson. 2017. *Trust is in the Keys of the Beholder: Extending SGX Autonomy and Anonymity*. Ph.D. Dissertation. Interdisciplinary Center, Herzliya.

[37] Luis Merino JP Aumasson. 2016. SGX Secure Enclaves in Practice Security and Crypto Review. https://www.blackhat.com/docs/us-16/materials/us-16-Aumasson-SGX-Secure-Enclaves-In-Practice-Security-And-Crypto-Review.pdf. (2016).

[38] Ranjit Kumaresan and Iddo Bentov. 2014. How to use bitcoin to incentivize correct computations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 30–41.

[39] Ranjit Kumaresan and Iddo Bentov. 2016. Amortizing secure computation with penalties. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 418–429.

[40] Joshua Lind, Ittay Eyal, Peter Pietzuch, and Emin Gün Sirer. 2016. Teechan: Payment channels using trusted execution environments. https://arxiv.org/pdf/1612.07766. pdf, (2016).

[41] Thomas Lundqvist, Andreas de Blanche, and H Robert H Andersson. 2017. Thing-to-thing electricity micro payments using blockchain technology. In *Global Internet of Things Summit (GIoTS), 2017*. IEEE, 1–6.

[42] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[43] Patrick Nelson. 2016.. Just one autonomous car will use 4,000 GB of data/day. (December 2016). http://www.networkworld.com/article/3147892/internet/ one-autonomous-car-will-use-4000-gb-of-dataday.html, (2016.).

[44] Mahdi Mahdavi Oliaiy, Mohammad Hassan Ameri, Javad Mohajeri, and Mohammad Reza Aref. 2017. A Verifiable Delegated Set Intersection without pairing. In *Electrical Engineering (ICEE), 2017 Iranian Conference on*. IEEE, 2047–2051.

[45] Henning Pagnia. 1999. *On the impossibility of fair exchange without a trusted third party*. Technical Report.

[46] Viet Pham, MHR Khouzani, and Carlos Cid. 2014. Optimal contracts for outsourced computation. In *International Conference on Decision and Game Theory for Security*. Springer, 79–98.

[47] Joseph Poon and Thaddeus Dryja. 2015. The bitcoin lightning network: Scalable off-chain instant payments. *Technical Report (draft)* (2015).

[48] The Golem Project. 2016. Golem Whitepaper. https://golem.network/doc/Golemwhitepaper.pdf. (2016).

[49] Joanna Rutkowska. 2016. Thoughts on Intel's upcoming Software Guard Extensions. http://theinvisiblethings.blogspot.co.uk/2013/08/thoughts-on-intels-upcoming-software.html. (2016).

[50] E. M. Schooler, D. Zage, J. Sedayao, H. Moustafa, A. Brown, and M. Ambrosin. 2017. An Architectural Vision for a Data-Centric IoT: Rethinking Things, Trust and Clouds. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 1717–1728. DOI: http://dx.doi.org/10.1109/ICDCS.2017.243

[51] Zihao Shan, Kui Ren, Marina Blanton, and Cong Wang. 2018. Practical Secure Computation Outsourcing: A Survey. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 31.

[52] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*.

[53] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *NDSS*.

[54] Jake Swearingen. 2018. http://nymag.com/selectall/2018/03/when-amazon-web-services-goes-down-so-does-a-lot-of-the-web.html. (2018).

[55] Doug Szajda, Barry Lawson, and Jason Owen. 2003. Hardening functions for large scale distributed computations. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*. IEEE, 216–224.

[56] Jelle van den Hooff, M Frans Kaashoek, and Nickolai Zeldovich. 2014. Versum: Verifiable computations over large public logs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1304–1316.

[57] Riad S Wahby, Max Howald, Siddharth Garg, Abhi Shelat, and Michael Walfish. 2016. Verifiable asics. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 759–778.

[58] Michael Walfish and Andrew J Blumberg. 2015. Verifying computations without reexecuting them. *Commun. ACM* 58, 2 (2015), 74–84.

[59] Johannes Winter. 2008. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*. ACM, 21–30.

[60] Gavin Wood. 2018. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf. (2018).

[61] ChongChong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and ChunXiao Xing. 2016. On the Performance of Intel SGX. In *Web Information Systems and Applications Conference, 2016 13th*. IEEE, 184–187.