Trust Trackers for Computation Offloading in Edge-Based IoT Networks

Matthew Bradbury
Department of Computer Science,
University of Warwick
M.Bradbury@warwick.ac.uk

Arshad Jhumka
Department of Computer Science,
University of Warwick
H.A.Jhumka@warwick.ac.uk

Tim Watson
WMG,
University of Warwick
tw@warwick.ac.uk

***Abstract*—Wireless Internet of Things (IoT) devices will be deployed to enable applications such as sensing and actuation. These devices are typically resource-constrained and are unable to perform resource-intensive computations. Therefore, these jobs need to be offloaded to resource-rich nodes at the edge of the IoT network for execution. However, the timeliness and correctness of edge nodes may not be trusted (such as during high network load or attack). In this paper, we look at the applicability of trust for successful offloading. Traditionally, trust is computed at the application level, with suitable mechanisms to adjust for factors such as recency. However, these do not work well in IoT networks due to resource constraints. We propose a novel device called Trust Tracker (denoted by Σ) that provides higher-level applications with up-to-date trust information of the resource-rich nodes. We prove impossibility results regarding computation offloading and show that Σ is necessary and sufficient for correct offloading. We show that, Σ cannot be implemented even in a synchronous network and we compute the probability of offloading to a bad node, which we show to be negligible when a majority of nodes are correct. We perform a small-scale deployment to demonstrate our approach.**

I. Introduction

As part of the Internet of Things (IoT), a large number of networked devices are expected to be deployed for a variety of purposes such as monitoring and actuation. The Cisco Annual Internet Report (2018–2023) predicts that by 2023 there are expected to be 14.7 billion machine-to-machine connections [1]. These include devices deployed in a variety of domains from healthcare [2], smart cities [3] and industrial applications [4]. Many of the IoT devices deployed will be resource-constrained with limited processing power, storage, energy availability, or a combination of these three and others.

Due to these resource constraints, it may be infeasible for computational and memory intensive tasks to be executed on the IoT device itself and they will instead need to be offloaded to more capable devices. Resource-rich devices that will process offloaded tasks include services accessed via the internet (e.g., a Cloud service) or via Edge nodes which are present in the same network as the resource-constrained devices. Certain tasks will have low latency requirements, meaning that Edge nodes become preferable over Cloud services.

When there exists multiple resource-rich Edge nodes, a problem that resource-constrained IoT nodes need to solve is which Edge node should a task be submitted to. This has typically been addressed [5, 6] by assessing a measure of *behavioural trust* that a resource-constrained node has in a resource-rich node to correctly execute a task correctly. A correctly executed task by a resource-rich node returns results that satisfy the user requirements. The evaluation of trust is typically performed at the application level. A node that requires trust information about other nodes will need to keep track of a history of its interactions with those nodes.

Unfortunately, such approaches do not scale in IoT networks due to the imposed resource constraints. Further, as trust is evaluated at the application level, interactions may be sporadic. As such, the trust models are enhanced with a decay model to model that relevancy of trust values may have decreased, i.e., to address the recency problem of the recorded interactions.

Existing trust-based evaluation approaches typically use a history of interactions to derive a trust value. For example, in the Beta Reputation System [7] interactions are classified as good or bad and the history of interactions is used to calculate the expected likelihood of a future event being good or bad. Logging interaction history is a reactive measure for trust evaluation. To circumvent the problem posed by resource constraints and interaction recency, in this work we propose a proactive approach for trust evaluation, which periodically challenges a resource-rich node with a non-trivial task.

To achieve this, we adapt the notion of failure detectors [8] for trust evaluation. We propose a novel device, called a Trust Tracker (denoted by Σ), that proactively records the nodes that can be trusted (or not). We investigate the power of Σ in solving the offloading problem and show equivalence between the two problems. In this paper, we make the following contributions:

1) We formalise the problem of trusted offloading and show that it is impossible in an asynchronous system.
2) We propose a powerful device namely a Trust Tracker Σ, and show that it is both necessary and sufficient to solve the trusted offloading problem.
3) We unfortunately show that it is impossible to implement Σ even in a synchronous system.
4) We propose a weaker specification for trusted offloading, that of probabilistic offloading problem, and show that the probability of offloading to an untrusted node is negligible in sufficiently large networks.

This work was supported by the PETRAS National Centre of Excellence for IoT Systems Cybersecurity [EPSRC Grant EP/S035362/1].

5) A demonstration is performed on a deployment of 6 wireless sensor nodes and 6 Raspberry Pis, showing that resource-constrained nodes are able to identify a misbehaving resource-rich node with high accuracy and select an appropriate alternative to offload tasks to.

The remainder of this paper is structured as follows. Related work on trust evaluation and task offloading is presented in Section II and the formalisation of the system and fault models in Section III. In Section IV the offloading problem and its impossibility condition is described. In Section V we present Trust Tracker to identify resource-rich nodes believed to be bad. We show that Trust Tracker cannot be implemented in a synchronous system and define a probabilistic version of the offloading problem in Section VI. In Section VII we describe the experimental setup used to gather results presented in Section VIII. Finally, we conclude with Section IX.

II. Related Work

A. Trust

Trust has different meanings in different contexts. For example, identity trust describes the ability for nodes to verify the authenticity of senders via "certificate-based frameworks" [9] which could be provided via message authentication codes or digital signatures. In this work we consider behaviour trust, which can be described as "the subjective belief, from the perspective of a *trustor* agent, that a *trustee* agent will act as they say they will do in a given context" [5, 7].

The Beta Reputation System (BRS) [7] uses the Beta distribution to model how a trustee is expected to behave based on past interactions classified as either good or bad. The expected value of the distribution can be used to predict the likelihood future event will be good. The BRS supports combining feedback from other agents, discounting unreliable feedback, and forgetting of older interactions which have become less relevant. Evaluation of trust is reactive based on previously observed interactions.

A limitation of the BRS is the assumption of stable system behaviour over time [10]. Forgetting (also called decay) can been used to weight recent events higher (e.g., with an exponentially weighted moving average [11]). However, for unstable systems (i.e., where behaviour changes frequently) this can lead to error in the prediction of the true system behaviour [12]. Alternatively, a finite state Hidden Markov Model (HMM) allows capturing dynamic behaviour of systems with lower error than Beta distribution-based trust models [10, 13]. Other approaches do not require a direct interaction with the trustee to form an opinion. Trust can be established from overhearing interactions between other agents [14].

However, there has been little work considering unreliability and resource constraints of nodes evaluating trust in general. A specific example, is the Collection Tree Protocol [15] for routing which use a combination of reactively assessing potentially unreliable communication links based on previously received messages and proactive assessment via beacons.

B. Task Offloading

In vehicular and cellular networks the task offloading problem is referred to as Multi-access Edge Computing (MEC) (previously Mobile Edge Computing) [16]. Whilst the scale of the compute may differ between highly resource-constrained IoT devices and vehicles/mobile phones, both involve a device that has insufficient knowledge or compute capability to execute a task so needs to offload it to a more capable device to perform the compute. A difference between highly resource-constrained IoT devices (such as embedded wireless sensor nodes) and devices associated with MEC is that the resource constraints of the IoT devices (e.g., 32 KiB of RAM) make expensive Edge evaluation techniques infeasible to implement.

A variety of different machine learning approaches have been used to evaluate trust. One approach is to frame trust evaluation as a classification problem. In [17] support vector machines were trained offline and used to facilitate behavioural trust classification. A downside is that a (potentially) large quality of labelled data may needed to train a model, which may not be available or representative of behaviour that will occur. Q-learning (a model-free reinforcement learning technique) was used to learn an offloading policy in [18] by providing a reward based on the action taken in states defined by the proposed system model. An advantage of this approach is training data is not required to design the reward function.

Linear programming (LP) has also been applied where the system is modelled as a LP and solvers are used to find an optimal allocation of tasks based on specified constraints. For example, [19] optimised according to bandwidth restrictions and [20] to minimise task latency. A downside is that LP is performed at one central location and needs global knowledge, requiring the cost of IoT nodes disseminating state.

Offloading can also be approached from a game theoretic viewpoint [21] where a plan is produced and typically a Nash equilibrium is sought. Games can involve non-cooperative or cooperative behaviour and be centralised or decentralised. As with LP, different performance metrics can be considered.

Different works have focused on optimising task offloading for different constraints such as bandwidth [19], latency [20], energy [22] and privacy [17]. Some of these works have framed offloading as a trust evaluation task, with different definitions of what a trustor is evaluating in a trustee. Therefore, there needs to be a clear understanding of what is being assessed by *trust* and how it is being assessed.

III. Models: System and Fault

In this section we present the system and fault models that will be used to describe the system and the ways in which it can fail. We then use these models to identify the properties of a strong trust detector.

A. System and Network Model

We assume a system that contains two classes of nodes:

- $V_R = \{r_1, r_2, \ldots, r_n\}$ is the set of resource-rich edge nodes that execute jobs.

- $V_C = \{c_1, c_2, \ldots, c_m\}$ is the set of resource-constrained IoT devices that submit jobs.
- A resource-rich node is not a resource-constrained node and vice versa $V_R \cap V_C = \emptyset$.
- The set of all nodes is given by $V = V_R \cup V_C$.

Thus, the network is modelled as a bipartite graph $G = (V_C, V_R, E)$, where $E \subseteq (V_C \times V_R)$. An edge (c, r) is in E if c can communicate with r and vice versa.

A system is said to be asynchronous if no bound is known on the message delivery time or the time between the execution of two consecutive instructions. It is synchronous if a bound is known for both. A system can be asynchronous, for example due to resource-constrained nodes sleeping because of duty cycling where bounding the sleep time is difficult. Another example could be that the network might be loaded, making bounding round trip time difficult. In an asynchronous system, we assume the existence of a fictitious clock device that returns the current time, to ease explanation. In a synchronous system we assume that the execution happens in well-defined rounds.

In this network the IoT devices are assumed to be highly resource-constrained, where devices may be similar to sensor nodes with limited CPU, RAM and energy storage. For example, the Zolertia RE-Mote [23] has a $32\,\mathrm{MHz}$ CPU, $32\,\mathrm{KiB}$ of RAM, a $800\,\mathrm{mAh}$ battery and optional support for an SD card (potentially meaning no stable storage). Communication in such devices is typically performed using IEEE 802.15.4, Bluetooth Low Energy or LoRaWAN.

We assume that there is an appropriate system which facilitates message encryption (if needed) and message authentication (such as via a message authentication code or digital signature) providing identity trust. We assume a task to be a program that requires computational and communication resources that are not available on an IoT device.

B. Process, Process Failure and Trust

In this paper, we assume nodes can behave arbitrarily, i.e., byzantine nodes [24]. This behaviour may have a malicious (e.g., network attacks) or benign cause (e.g., network load).

A failure pattern $\mathcal{F}$ is a function $\mathcal{F} : \mathcal{T} \to 2^{V_R}$, where $\mathcal{F}(t)$ denotes the set of resource-rich processes that are bad at time t. We say a resource-rich process r turns good at time t if $r \notin \mathcal{F}(t)$ and r is bad at time t if $r \in \mathcal{F}(t)$. We say that a resource-rich process r fails (or turns bad) at time t if r is good at time $t-1$ and r is bad at time t (i.e., $r \notin \mathcal{F}(t-1) \wedge r \in \mathcal{F}(t)$). We say that r recovers at time $t \geq 1$ if r is bad at time $t-1$ and r is good at time t (i.e., $r \in \mathcal{F}(t-1) \wedge r \notin \mathcal{F}(t)$). A process r can be classified (according to $\mathcal{F}$) as always-good, eventually-good, eventually-bad and unstable as follows:

Given a specification (or guarantee) $\P_r$ by r,

- Always Good: r always executes according to $\P_r$.
- Eventually Good: r can initially violate $\P_r$, but $\exists t > 0$ after which r is always good.
- Eventually Bad: $\exists t > 0$ after which r always violates $\P_r$.
- Unstable: r alternates between being violating and executing according to $\P_r$.

Definition III.1 (Trusted)**.** A process r is trusted if it always good or is eventually good in a finite amount of time. A process is untrusted if it is not trusted.

We denote the set of trustworthy, untrustworthy and unstable processes in $\mathcal{F}$ by $\textsc{trust}(\mathcal{F})$, $\textsc{untrust}(\mathcal{F})$ and $\textsc{unstable}(\mathcal{F})$ respectively.

C. Failure and Trust Detectors

A failure detector is a (software) device that is responsible for the detection of node crashes in a distributed system [25]. A higher level application will query the failure detector to obtain failure information.

Similarly, we define a device called a trust detector that can be queried at time $t \in \mathcal{T}$, which returns the set of processes it suspects to be bad at t. A trust detector history $\mathcal{H} : V_C \times \mathcal{T} \to 2^{V_R}$ denotes the value of the trust detector for process c at time t, i.e., if the trust detector at a resource-constrained process c is queried at time t, then $\mathcal{H}(c, t)$ contains the set of resource-rich processes that c suspects to be bad at time t.

A trust detector $\mathcal{D} : \mathcal{F} \to 2^{\mathcal{H}}$ maps a failure pattern $\mathcal{F}$ to a set of trust detector histories, i.e., $\mathcal{D}(\mathcal{F})$ returns the the set of possible trust detector histories permitted by $\mathcal{D}$ for $\mathcal{F}$. However, not all possible trust detector histories are useful. For example, trust detector histories where node failures are not detected are not useful. There is not just one type of trust detector, but many possible types depending on what properties they provide, that is, depending on their strengths.

To reason on the usefulness of a trust detector history, we generally require them to satisfy certain properties, similar to failure detectors [25]. The completeness property of a trust detector captures its ability to detect correct node failures and the accuracy property captures the ability to avoid wrong suspicions. Due to the uncertainty in an asynchronous system, these properties cannot be taken for granted, giving rise to different degrees of completeness and accuracy. For example, in an IoT network, uncertainty can arise due to wireless interference or network load. In such a case, a trust detector may report that a node has failed when it is actually correct.

Definition III.2 (Perfect Trust Detector)**.** A perfect trust detector has the following two properties:

- Strong accuracy: No resource-rich process r is suspected by any resource-constrained process c before it fails.
  $\forall \mathcal{F}, \forall \mathcal{H} \in \mathcal{D}(\mathcal{F}), \forall t \in \mathcal{T}, \forall r \in V_R, r \notin \mathcal{F}(t), \forall c \in V_C \cdot r \notin \mathcal{H}(c, t)$.
- Strong completeness: Every resource-rich process r that fails is eventually permanently suspected by all resource-constrained process c.
  $\forall \mathcal{F}, \forall \mathcal{H} \in \mathcal{D}(\mathcal{F}), \forall r \in V_R, r \in \textsc{untrust}(\mathcal{F}), \forall c \in V_C, \exists t \in \mathcal{T}, \forall t' \geq t \cdot r \in \mathcal{H}(c, t')$.

We call a trust detector that satisfies both the strong completeness property and the strong accuracy property a *perfect trust detector*. The accuracy property refers to the ability of the trust detector to avoid incorrect suspicions while the completeness property addresses the ability of the trust detector to detect failing (resource-rich) nodes.

D. Algorithms and Computation

Chandra and Toueg [8] define a computation (or execution) to be a tuple $C = (\mathcal{F}, \mathcal{D}, \mathcal{I}, \mathcal{S}, \mathcal{T})$ where $\mathcal{F}$ is a failure pattern, $\mathcal{D}$ is a failure[1] detector, $\mathcal{I}$ is the system's initial state, $\mathcal{S}$ is a sequence of algorithm steps, and $\mathcal{T}$ is a sequence of increasing time values when the algorithm steps are taken. In this paper, we will study algorithms that make use of devices similar to failure detectors. However, our definition of computation will be slightly different [26], but equivalent to that of Chandra and Toueg [8]. We define two functions: a step function A_s from $\mathcal{T}$ to the set of all algorithm steps that occurred at that time, and a process function A_p from $\mathcal{T}$ to V. In other words, function $A_p(t)$ denotes the process that takes a step at time t and $A_s(t)$ identifies the step that was taken. If no process takes a step at time t, both the step function and the process function evaluate to $\perp$. Without loss of generality, we assume at most one process taking a step at any one time.

To account for the possibility of misbehaviours, we augment our notion of computation with a further set F_{tr} of (faulty) steps to become possible. The actions in F_{tr} model (resource-rich) node failures [27] when they become Byzantine. A computation C' in the presence of faults (or a faulty computation) is thus given as $C' = (\mathcal{F}, \mathcal{D}, \mathcal{I}, A'_s, A_p)$, where $A'_s(t)$ returns the step, possibly faulty, taken by a process at time t and $A_p(t)$ returns the process that takes the step at time t.

A specification is a set of computations. A program P satisfies a specification ¶ if every computation of P is in ¶. Alpern and Schneider [28] state that every specification can be described as the conjunction of a safety and liveness property. A safety property states that something bad should not happen; a liveness property states that something good will eventually happen. Formally, the safety property identifies a set of finite computation prefixes that should not appear in any computation. A liveness property identifies a set of computation suffixes that every computation should include.

IV. Offloading Problem

The offloading problem can be explained as follows: A node $c \in V_C$ has a job j to execute. The computational requirements of j cannot be met by node c, therefore, c needs to find a resource-rich node $r \in V_R$ to execute j on its behalf. We say that a node c offloads a job j to a node r when node r sends a special control message $\text{OFFLOAD}\langle j \rangle$ from c. Thus, if c sends $\text{OFFLOAD}\langle j \rangle$ to r at time t_0, then r will receive $\text{OFFLOAD}\langle j \rangle$ at a time t_1 where $t_1 > t_0$.

Definition IV.1 (Offloading Problem)**.** Given a resource-constrained node $c \in V_C$ and a task j, the offloading problem of j by c is defined as follows: There exists a resource-rich node $r \in V_R$ such that:

- Correctness: c offloads j to r only if c trusts r.
- Trust: Eventually, c trusts r permanently[2].

[1] In this paper, it will represent a trust detector.

[2] Permanently (in this setting) does not mean forever after some point, but long enough for the system to make progress.

We define a software device, which we call an *offloading engine* and which we denote by $\mathcal{O}$, that is responsible for job offloading. The properties of $\mathcal{O}$ are thus as follows:

- Safety: $\mathcal{O}$ only returns a set $T_r \subseteq V_r$ of trusted nodes, i.e., $\forall \mathcal{F}, T_r \subseteq \text{TRUST}(\mathcal{F})$.
- Liveness: Eventually, $\mathcal{O}$ returns a set $T_r \subseteq V_r$ of resource-rich nodes.

Once the set T_r is available, then $\mathcal{O}$ chooses an edge node from that list. We now present our first important result.

Theorem 1 (Impossibility of Offloading)**.** *Given an asynchronous network $G = (V, A)$, $V = V_C \cup V_R$ where all nodes $c \in V_C$ are equipped with a perfect trust detector, a job j, then it is impossible to solve the offloading problem of j by c.*

Proof. We assume such a deterministic offloading algorithm $\mathcal{A}$ exists and then show a contradiction.

Consider a failure pattern $\mathcal{F}_0$ where there are no failures and consider a computation $E_0 = (\mathcal{F}_0, \mathcal{D}, \mathcal{I}, A_s, A_p)$ which occurs as follows. At time t_0, a node c has a job j to offload. Since $\mathcal{A}$ is correct and there are no failures, the trust detector at c returns $\emptyset$ (at all times) and c eventually offloads j onto a node r at time $t_1 > t_0$ and r receives $\text{OFFLOAD}\langle j \rangle$ at $t_2 > t_1$.

Now, consider a different failure pattern $\mathcal{F}_1$ where a set $B_r \subset V_R$ are untrusted in $\mathcal{F}_1$. Assume that $\mathcal{F}_1$ is identical to $\mathcal{F}_0$ until t_1. Assume that the node r becomes bad at time t_1, i.e., $r \in \mathcal{F}_1(t_1)$. Now, consider an computation E_1 which is identical to E_0 until t_1. Since $\mathcal{A}$ is deterministic, $\mathcal{A}$ will choose r for offloading. However, as r turns bad at t_1, $\mathcal{A}$ needs to offload before t_1 else it will violate the correctness problem of the offloading problem. This means that the system needs to be synchronous to meet this time bound, which contradicts our assumption of an asynchronous system. □

The issues here are that resource-rich nodes can be unstable and the system being asynchronous. It can be observed that even if the system is synchronous, unstable nodes will always be a problem. Thus, we need a device which is strictly stronger than a perfect trust detector to solve the offloading problem.

V. Σ: A Trust Tracker Device

A perfect trust detector returns a list of nodes it suspects to be bad. Unfortunately, the inability to discriminate between a bad node and an unstable node can lead to violations. As such, we propose a new device, called a *trust tracker*, that returns a list of nodes believed to be bad and also an epoch vector $Ep : V_R \to \mathbb{N}_0$ which captures the number of times the trust tracker believes a resource-rich node has failed and recovered (and vice versa).

Thus, the trust tracker device has the following properties:

- Completeness: for all bad resource-rich processes $r \in V_R$ and for all process $c \in V_C$, either c eventually permanently suspects r or r's epoch number at c is unbounded.
- Accuracy: For some good resource-rich process $r \in V_R$ and for every process $c \in V_C$, eventually c permanently trusts r and r's epoch number at c stops changing.

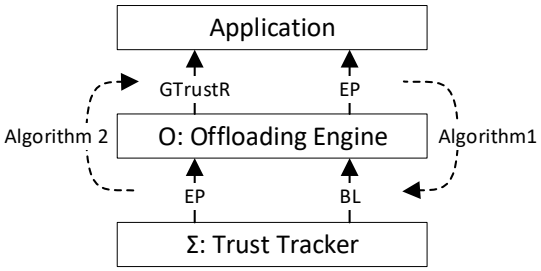


Figure 1: Architecture for sufficiency and necessity proofs.

The completeness property of the trust detector captures issues caused by untrusted nodes (i.e., bad and unstable nodes). It associates edge nodes with an epoch number to detect unstable nodes. On the other hand, the accuracy property stipulates that (eventually) good nodes are permanently trusted. We denote the trust tracker device by Σ. At this point, we wish to determine the role played by Σ in the offloading problem. We now present our second main result.

Theorem 2 (Necessity and Sufficiency of Σ)**.** *Given a synchronous system $G = (V_R \cup V_C, A)$, it is possible to solve the offloading problem if and only if each node $c \in V_C$ is equipped with Σ, i.e., $\mathcal{O}$ and Σ are equivalent.*

Proof.

- **Sufficiency**: To prove that Σ is sufficient to solve the offloading problem, we provide an algorithm (see Algorithm 1) that uses Σ to construct $\mathcal{O}$.
- **Necessity**: To prove the necessity of Σ to solve the offloading problem, we develop an algorithm (see Algorithm 2) that emulates Σ using the output of $\mathcal{O}$. □

The relation between $\mathcal{O}$ and Σ is shown in Figure 1.

A. Sufficiency of Σ

Algorithm 1 shows how to transform the output of Σ (Line 6) to obtain the output of $\mathcal{O}$, the offloading engine, as represented in Figure 1. Line 6 (of Algorithm 1) represents the input which Σ submits to $\mathcal{O}$. At the start of every synchronous round, Σ samples each resource-rich node. As it receives messages from resource-rich nodes, it updates its list of bad nodes (BL) and the list of unstable nodes (via the epoch vector) and outputs it to $\mathcal{O}$. We assume a threshold σ to identify unstable nodes. When $\mathcal{O}$ is ready to select an edge node for offloading, Σ outputs the list $GtrustR$ of trusted nodes (those that are good or eventually good), as well as the epoch vector.

B. Necessity of Σ

To prove the necessity of Σ to solve the offloading problem, Algorithm 2 shows how to transform the output of the offloading engine $\mathcal{O}$ (see Figure 1) to obtain the output of Σ, the trust tracker device. The offloading engine $\mathcal{O}$ returns a list of trusted nodes and the epoch vector. The output of $\mathcal{O}$ is (T_r, E), which are then inputs to Algorithm 2. Using the epoch vector, Algorithm 2 computes the set of unstable nodes and, using the set of trusted nodes T_r, it then computes the set BL of bad nodes. The algorithm then outputs (BL, E) for Σ.

Algorithm 1 Sufficiency: Σ sufficient to solve $\mathcal{O}$

▷ Output of Σ to obtain offloading engine
1: **function** INIT
2: START(off, δ) ▷ Timer
3: $Ep \leftarrow V_R \times \{0\}$ ▷ $Ep \in V_R \to \mathbb{N}_0$
▷ Trusted nodes in one round
4: $trustR \leftarrow V_R$ ▷ $trustR \in 2^{V_R}$
▷ Those identified as trusted over multiple rounds
5: $GtrustR \leftarrow V_R$ ▷ $GtrustR \in 2^{V_R}$
6: **event** EVALUATED$(BL, E) \rightarrow$ ▷ BL: bad list, E: Epoch
7: Ep, $trustR \leftarrow E$, V_R ▷ Reset
8: $UR \leftarrow \{u \mid u \in V_R \wedge E(u) \geq \sigma\}$ ▷ Unstable nodes
9: $trustR \leftarrow trustR \setminus BL$ ▷ Remove bad nodes
10: $trustR \leftarrow trustR \setminus UR$ ▷ Remove unstable nodes
▷ Nodes that are permanently good
11: $GtrustR \leftarrow GtrustR \cap trustR$
12: **timeout** $(off) \rightarrow$ ▷ Offloading decision time
13: **if** $GtrustR \neq \emptyset$ **then**
14: **return** $(GtrustR, Ep)$
15: **else**
16: START(off, δ)
17: $GtrustR \leftarrow trustR$

Algorithm 2 Necessity: Σ necessary to solve $\mathcal{O}$

▷ Output of offloading engine to obtain Σ
1: **function** OUTPUT(T_r, E)
2: $UR \leftarrow \{u \mid u \in V_R \wedge E(u) \geq \sigma\}$
3: $BL \leftarrow V_R \setminus (UR \cup T_r)$
4: **return** (BL, E)

VI. Implementing Σ and Probabilistic Offloading

The use of Σ makes it possible to solve the offloading problem, however, it still needs to be understood if it is possible to implement Σ in a synchronous system. Unfortunately, the answer is negative. Our third major result is thus:

Theorem 3 (Impossibility of implementing Σ)**.** *Given a synchronous system, it is impossible to implement Σ.*

Proof. We assume an algorithm $\mathcal{A}$ exists for Σ and then show a contradiction.

Consider a failure pattern $\mathcal{F}_0$ where there are no failures and consider a computation $E_0 = (\mathcal{F}_0, \mathcal{D}, \mathcal{I}, A_s, A_p)$ of Σ which occurs as follows: At time t_0 (start of a synchronous round), Σ collects information about all nodes $r \in V_R$ and returns information at the end of the round. Since $\mathcal{A}$ is correct and there is no failure, Σ will return $(\emptyset, V_R \times \{0\})$.

Consider a failure pattern $\mathcal{F}_1$ where there is a failure of a single node r sometime during the first synchronous round and turns good before the end of the round. Consider a computation of Σ $E_1 = (\mathcal{F}_1, \mathcal{D}, \mathcal{I}, A_s, A_p)$. Since E_0 and E_1 are indistinguishable for Σ, Σ will return $(\emptyset, V_R \times \{0\})$. This contradicts the fact that there has been a failure in E_1. □

Because of the impossibility of implementing Σ, we require

resource-constrained processes to test edge nodes at regular intervals. To increase the chance of detecting every failure, the sampling periods at the sensor processes should be set sufficiently small so that all edge node failures can be detected. Because fault occurrences are random, even arbitrarily small sampling period may miss faults.

Therefore, we introduce a weaker version of the offloading problem, that of probabilistic offloading. The probabilistic version of the offloading problem is a weaker version of the more general problem in that it only requires the offloading problem to be satisfied with high probability, i.e., it allows the offloading problem to fail.

Definition VI.1 (Probabilistic Offloading Problem)**.** Given a resource-constrained node $c \in V_C$ and a task j, the offloading problem of j by c is defined as follows: There exists a resource-rich node $r \in V_R$ such that:

- Correctness: c offloads j to r only if c trusts r with high probability p_c.
- Trust: Eventually, c trusts r permanently with high probability p_t.

To calculate the value of p_c and p_t, we need to understand the probability of Σ to return incorrect information to $\mathcal{O}$.

We now compute the value of p_c, the probability that the correctness condition of offloading is not violated. For the violation to happen the output variable $GtrustR$ needs to contain at least one node that keeps failing but cannot be detected.

The main factor that will affect p_c is when an unstable node appears to be good. This happens when a node turns bad shortly after being sampled but turns good before it is next sampled. We refer to such a node as a *fake* node.

Denote by p the probability of a node to be fake in a synchronous round of Σ. The expected number of fake nodes in any round is thus pR, where $R = |V_R|$. Let F be a random variable that denotes the number of fake nodes in a single round. We assume F to follow a Poisson distribution.

$$F \sim Po(\lambda) \text{ where } \lambda = pR \tag{1}$$

The probability of at least one fake node in a round is:

$$p_{\geq 1} = \Pr(F \geq 1) = \Pr(F \neq 0) = 1 - \exp(-pR) \tag{2}$$

Denote by σ, the number of rounds before an offloading decision is made. The probability that the same node is fake over the duration before offloading (p_+) is less than or equal to the probability that at least one node is fake for σ rounds:

$$p_+ \leq (p_{\geq 1})^\sigma = (1 - \exp(-pR))^\sigma \tag{3}$$

Since $|GtrustR| \geq 1$, it means that the probability of choosing a fake node (p_v) is:

$$p_v \leq \frac{p_+}{|GtrustR|} = \frac{(1 - \exp(-pR))^\sigma}{|GtrustR|} \tag{4}$$

Then, the probability that the correctness condition of offloading is not violated is given by $p_c \geq 1 - p_v$. As $|GtrustR|$ increases $p_c \to 1$. The graphs in Figure 2 show the value of p_c for various parameterisations. When there are a large number of trusted nodes (such as when $|GTrustR| = {}^{R}/_{2}$) there is a low probability of offloading to a fake node. When there is a small number of trusted nodes, the probability of offloading to a fake node significantly increases as the probability that a resource-rich node is fake (p) increases.

Algorithm 3 Algorithm for Σ based on Challenge-response from a resource-constrained node. Protocol at a resource-constrained node c.

```
1:  cs ← ∅                                   ▷ Mapping of V_R to challenges
2:  challenge-response ← ∅                   ▷ Mapping of V_R to timers
3:  BL ← ∅                                   ▷ Set of bad rr nodes
4:  E ← V_R × {0}                            ▷ Epoch number for each rr node
5:  function INIT
6:      START(challenge-timer, τ)
7:  timeout (challenge-timer) →
8:      for v_r ∈ V_R do
9:          cs[v_r] ← CREATECHALLENGE()
10:         send Challenge⟨cs[v_r]⟩ to v_r
11:         START(challenge-response[v_r], Δ) ▷ Start timeout
12:     START(challenge-timer, τ)
13: receive ChallengeResponse⟨cr⟩ from v_r →
14:     if ISRUNNING(challenge-response[v_r]) then
15:         STOP(challenge-response[v_r])
16:         if VALIDATE(cs[v_r], cr) then
17:             EVALUATE(v_r, Succeeded)
18:         else
19:             EVALUATE(v_r, Failed)
20: timeout (challenge-response[v_r]) →
21:     EVALUATE(v_r, Timeout)
22: function EVALUATE(v_r, res)
23:     if res = Succeeded then
24:         if v_r ∈ BL then                 ▷ Was previously bad
25:             E[v_r] ← E[v_r] + 1
26:             BL ← BL \ {v_r}              ▷ Remove from bad list
27:     else
28:         if v_r ∉ BL then                 ▷ Was previously good
29:             E[v_r] ← E[v_r] + 1
30:             BL ← BL ∪ {v_r}              ▷ Add to bad list
31:     signal EVALUATED(BL, E)              ▷ To Algorithm 1 Line 6
```

Algorithm 4 Algorithm for Σ based on Challenge-response from a resource-constrained node. Protocol at a resource-rich node r.

```
1:  cq ← []          ▷ cq is a queue of challenges and V_C nodes
2:  receive Challenge⟨c⟩ from v_c →
3:      cq ← cq ⌢ (c, v_c)                   ▷ ⌢ to append to queue
4:  event DOCHALLENGE :: cq ≠ [] →
5:      (c, v_c) ← POP(cq)
6:      cr ← COMPUTERESPONSE(c)
7:      send ChallengeResponse⟨cr⟩ to v_c
```

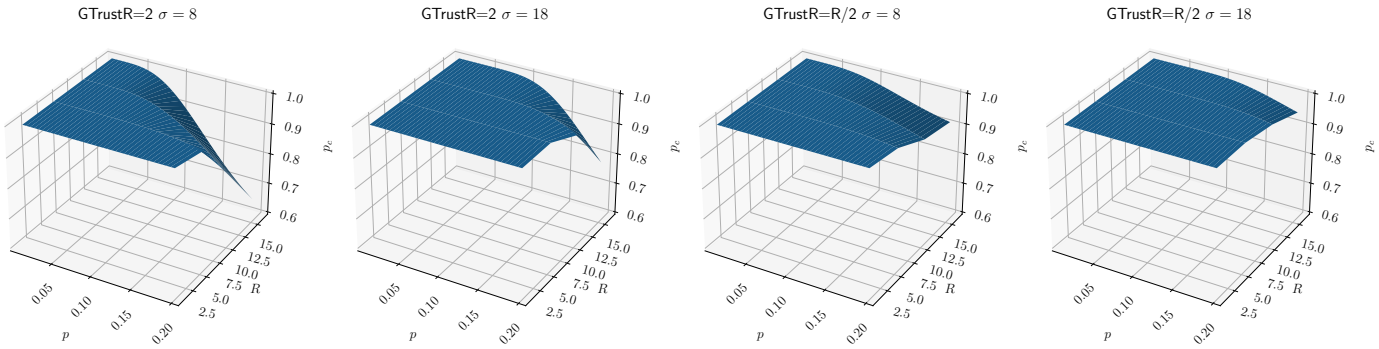


Figure 2: The probability of a correct offload (p_c) when varying: the number of resource-rich nodes (R), the probability of a resource-rich node being fake (p), the number of samples performed (σ), and the number of trustworthy nodes ($|GTrustR|$).

VII. Experimental Setup

A small deployment of 6 Zolertia RE-Mote Rev.B were used to perform testing, with these roles: 1 as the root, 2 and 6 as resource-rich Edge nodes, and 3, 4 and 5 as resource-constrained IoT nodes. All the RE-Motes were present within each other's communication range and were connected to their own Raspberry Pi to log output. The Raspberry Pi connected to the root node provided network services (such as Edge node discovery) and the Raspberry Pis connected to the Edge nodes ran the Edge applications which required additional compute capability. A sample application was implemented using [29] and performed periodic environment monitoring every $1\,\text{min}$.

A simple challenge and response was implemented similar to proof-of-work mining in blockchain [30]:

1) The resource-constrained node generates a random 32 bytes b, chooses a difficulty d and a deadline t and sends this to the resource-rich node.
2) The resource-rich node searches for a prefix p to b such that the first d bytes of SHA256($p\|b$) are 0.
3) If the resource-rich node finds such a prefix within the deadline t it returns that prefix to the resource-constrained node, otherwise returns an empty byte string.
4) On receipt of the prefix, the resource-constrained node validates the response by checking that the first d bytes of SHA256($p\|b$) are 0.

In these experiments, the difficulty d was 2 and the deadline t was $40\,\text{s}$. If a correct response was received after the deadline it was not considered a positive trust evaluation. A challenge was sent every $2\,\text{min}$ to one resource-rich node and then moves to the next resource rich node in a circular list. In this scenario a challenge was sent to a resource-rich node every $4\,\text{min}$.

Three sets of experiments were performed; (i) both resource-rich nodes are permanently good, (ii) one resource-rich node is permanently good and the other is permanently bad, and (iii) one resource-rich node is permanently good and the other is unstable. When a resource-rich node is unstable it switches between acting good and bad every $20\,\text{min}$. When acting good it correctly executes and responds to challenges. When acting bad the resource-rich node chooses randomly between sending an invalid response or not sending a response.

VIII. Results

A. Cost of Executing Challenge

The graphs in Figure 3 show the cost of executing the challenge. Both nodes were good, i.e., they correctly execute the challenge and returned a response. The average number of iterations to find a solution was $72\,000 \pm 11\,700$ (3 sf) with an average execution duration of $1.52\,\text{s} \pm 0.25\,\text{s}$ (3 sf). These values include 95% confidence intervals. Figure 3b shows that the challenge generation does not greatly bias the difficulty of the generated jobs submitted to different resource-rich nodes.

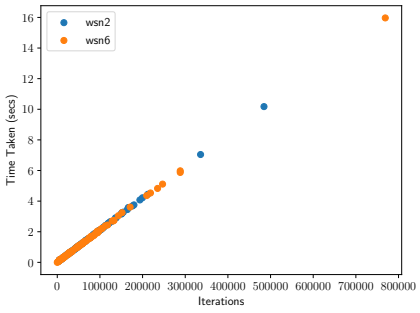
B. Two Good Resource-Rich Nodes

When both resource-rich nodes behave well there are few events that cause a resource-constrained node to lose trust in that resource-rich node. Two events that could happen are due to network unreliability, where an acknowledgement of receipt of a task fails to be received at the resource-constrained node or the completed task fails to be delivered. These errors are respectively named `NO_ACK` and `TIMEOUT`. When a loss of trust occurs it is represented by a grey region, when a resource-rich node is trusted it is indicated by a green region.

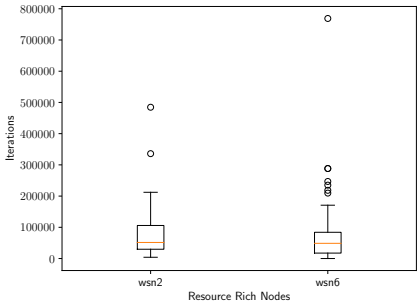
As can be seen in Figure 4 there are a few events that cause temporary loss of trust in a resource-rich node. These bad events cause the epoch number to increase, as does the subsequent recovery. Without network unreliability issues these instances would not have been observed. However, these results show that the resource-constrained nodes are able to recover from these transient network failures.

C. One Good and One Bad Resource-Rich Node

When there is one resource-rich node that is always good and another that is always bad, the resource-constrained nodes are able to classify them as such as shown in Figure 5. Once a node is classified as untrustworthy it is not possible for a resource-constrained node to ever change that opinion as the resource-rich node will never produce a valid challenge response in the future. As before, nodes that always behave well can become suspect due to network unreliability, but the resource-constrained nodes can eventually recover. An issue here is that due to a lack of redundancy in the resource-rich nodes available, a resource-constrained node may have no

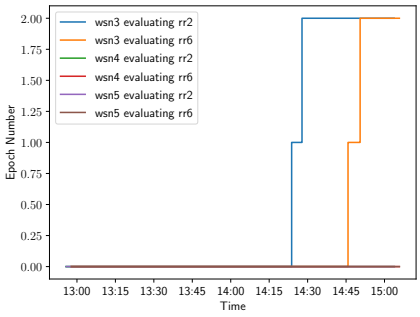


(a) The number of prefixes searched to find a solution versus the time taken.

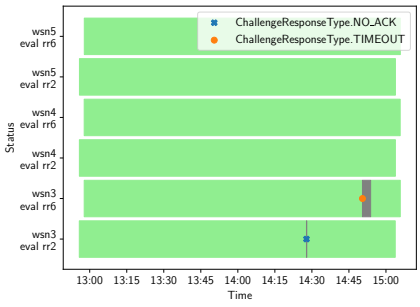


(b) A comparison between the load caused by the challenge on two different resource-rich nodes.

Figure 3: Challenge performance when both resource-rich nodes are good.

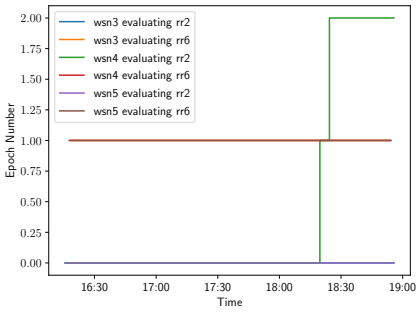


(a) Evolution of the Epoch number over time.

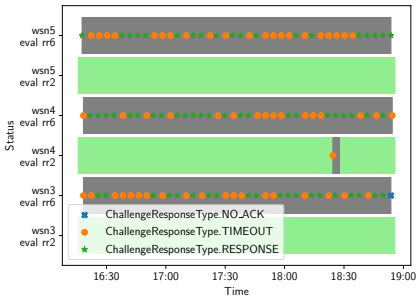


(b) Times at which resource-constrained nodes trusted resource-rich node. Events that led to loss of trust are indicated.

Figure 4: Results for when both resource-rich nodes 2 and 6 are good.



(a) Evolution of the Epoch number over time.



(b) Times at which resource-constrained nodes trusted a resource-rich node. Events that led to loss of trust are indicated.

Figure 5: Results for when resource-rich node 2 is good and 6 is bad.

suitable target to offload jobs to. Networks should be suitably provisioned with sufficient resource-rich nodes such that the likelihood of there being insufficient nodes to offload to is low.

In addition to the previous errors that lead to a resource-rich node being added to the bad list, the `RESPONSE` error is also present. This occurs when a resource-rich node returns a result that does not successfully complete the challenge.

D. One Good and One Unstable Resource-Rich Node

Figure 6 shows results for when resource-rich node rr2 was always stable and rr6 was unstable. Figure 6c shows that while rr6 was bad, fewer jobs were offloaded to it and instead rr2 was preferred. This is because the wsn nodes were able to detect these periods of bad behaviour as shown by the true negative time blocks in Figure 6d. In general there is a period of falsely detecting a resource-rich node as good, before detecting it as bad, followed by a period of falsely detecting it as bad. This is caused by the rate at which challenges are sent to the resource-rich nodes leading to a period of out-of-date evaluation. The confusion matrices in Table I show the accuracy of trust evaluation. There is a high accuracy of identifying the correct behaviour of the good rr2 node (>98%) and a lower but still high accuracy (84%) in the unstable rr6 node.

It is important to consider the relation between the rate at which challenges are sent, the deadline for the challenge, and the duration that resource-rich nodes remain stable for. When

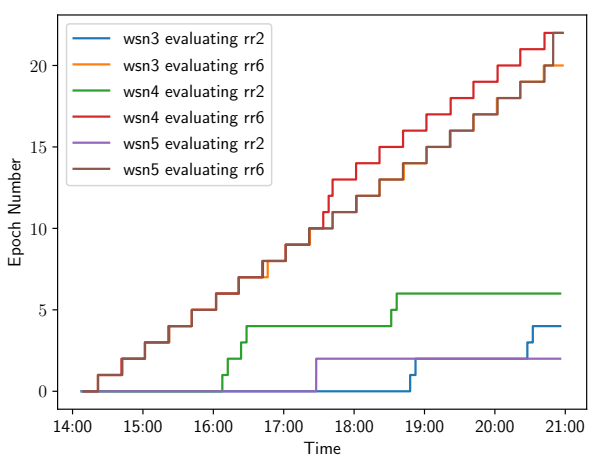
	wsn3		wsn4		wsn5	
	T	U	T	U	T	U
rr2 AG	0.98	0.02	0.98	0.02	0.99	0.01
rr2 AB	0.0	0.0	0.0	0.0	0.0	0.0
rr6 AG	0.43	0.08	0.43	0.08	0.43	0.08
rr6 AB	0.08	0.41	0.08	0.41	0.08	0.41

Table I: Error matrices showing the percentage of time resource-constrained nodes (wsn) considered resource-rich nodes (rr) as being trusted or not. T = trusted, U = untrusted, AG = actually good, AB = actually bad.

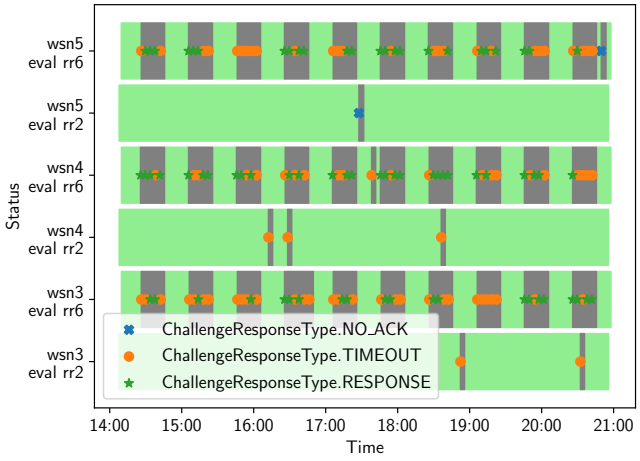
behaviour changes very frequently, the time between challenges and the duration allowed for the challenge needs to be small in order to minimise resource-constrained nodes incorrectly evaluating the status of a resource-rich node. However, this causes a high number of challenges to be sent, so a trade-off needs to be made between accuracy and challenge frequency.

IX. Conclusion

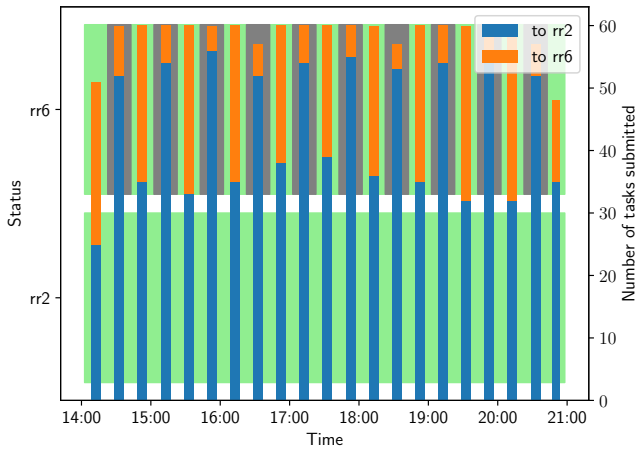
In this paper, we have proposed a novel device called Σ to capture trusted resource-rich nodes in an edge-based IoT networks. We have shown the power of Σ in solving the offloading problem. Our small scale deployment shows the applicability of our approach. The main novelty is our approach targets the lower level of the network stack, to provide trust



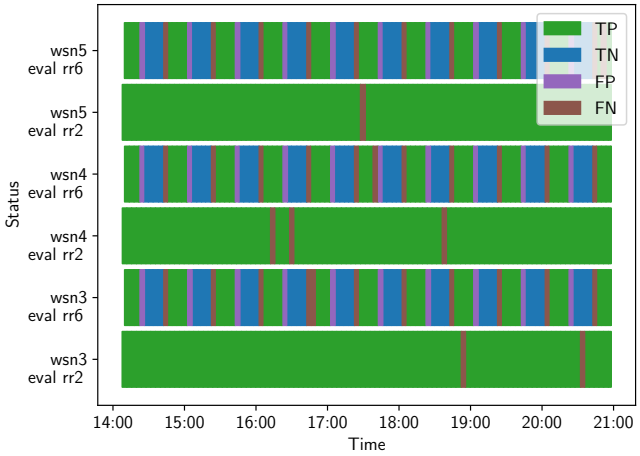
(a) Evolution of the Epoch number over time.



(b) Times at which resource-constrained nodes trusted resource-rich nodes. Events that led to loss of trust are indicated.



(c) The true status of resource-rich nodes and the number of tasks submitted to them in a time window where their behaviour was stable.



(d) Was the trust correctly evaluated? TP = trusted when good, TN = untrusted when bad, FP = trusted when bad, FN = untrusted when good.

Figure 6: Results for when resource-rich node 2 is good and 6 is unstable.

information to higher level applications. This approach makes it suitable for memory constrained IoT networks as extensive interaction histories are not logged and a decay model is eschewed in favour of a proactive mechanism.

One limitation to our approach is that it only evaluates trust in a resource-rich node based on its response to the submitted challenges. This means that a resource-rich node could correctly respond to a challenge, but then incorrectly respond to a per-application task. Therefore, in future work we will investigate a hybrid system of proactive challenges and reactive event gathering to build a more accurate trust evaluator. We will also investigate issues of period adjustments for energy efficiency.

Data Statement

The software used to generate these results can be found at https://github.com/MBradbury/iot-trust-task-alloc. The data gathered and presented in this paper can be found at [31].

References

[1] Cisco, "Cisco Annual Internet Report (2018–2023)," San Jose, CA, USA, White Paper C11-741490-01, Mar. 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf

[2] H. Habibzadeh, K. Dinesh, O. Rajabi Shishvan, A. Boggio-Dandry, G. Sharma, and T. Soyata, "A Survey of Healthcare Internet of Things (HIoT): A Clinical Perspective," *IEEE Internet of Things Journal*, vol. 7, no. 1, pp. 53–71, 2020.

[3] L. U. Khan, I. Yaqoob, N. H. Tran, S. M. A. Kazmi, T. N. Dang, and C. S. Hong, "Edge Computing Enabled Smart Cities: A Comprehensive Survey," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 10 200–10 232, 2020.

[4] T. Qiu, J. Chi, X. Zhou, Z. Ning, M. Atiquzzaman, and D. O. Wu, "Edge Computing in Industrial Internet of

Things: Architecture, Advances and Challenges," *IEEE Communications Surveys Tutorials*, pp. 1–1, 2020.

[5] P. Taylor, L. Barakat, S. Miles, and N. Griffiths, "Reputation assessment: a review and unifying abstraction," *The Knowledge Engineering Review*, vol. 33, p. e6, 2018.

[6] H. Yu, Z. Shen, C. Leung, C. Miao, and V. R. Lesser, "A Survey of Multi-Agent Trust Management Systems," *IEEE Access*, vol. 1, pp. 35–50, 2013.

[7] A. Jøsang and R. Ismail, "The Beta Reputation System," in *15th Bled Electronic Commerce Conference*. Bled, Slovenia: University of Maribor Press, 17–19th June 2002.

[8] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[9] E. Aivaloglou, S. Gritzalis, and C. Skianis, "Trust Establishment in Ad Hoc and Sensor Networks," in *Critical Information Infrastructures Security*, J. Lopez, Ed. Berlin, Heidelberg: Springer, 2006, pp. 179–194.

[10] E. ElSalamouny, V. Sassone, and M. Nielsen, "HMM-Based Trust Model," in *Formal Aspects in Security and Trust*, P. Degano and J. D. Guttman, Eds. Berlin, Heidelberg: Springer, 2010, pp. 21–35.

[11] Y. Wang, I. Chen, J. Cho, and J. J. P. Tsai, "Trust-Based Task Assignment With Multiobjective Optimization in Service-Oriented Ad Hoc Networks," *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 217–232, 2017.

[12] E. ElSalamouny, K. T. Krukow, and V. Sassone, "An analysis of the exponential decay principle in probabilistic trust models," *Theoretical Computer Science*, vol. 410, no. 41, pp. 4067–4084, 2009.

[13] E. ElSalamouny and V. Sassone, "An HMM-Based Reputation Model," in *Advances in Security of Information and Communication Networks*, A. I. Awad, A. E. Hassanien, and K. Baba, Eds. Berlin, Heidelberg: Springer, 2013, pp. 111–121.

[14] M. J. Probst and Sneha Kumar Kasera, "Statistical trust establishment in wireless sensor networks," in *2007 International Conference on Parallel and Distributed Systems*, 2007, pp. 1–8.

[15] O. Gnawali, R. Fonseca, K. Jamieson, M. Kazandjieva, D. Moss, and P. Levis, "CTP: An Efficient, Robust, and Reliable Collection Tree Protocol for Wireless Sensor Networks," *ACM Trans. Sen. Netw.*, vol. 10, no. 1, pp. 16:1–16:49, Dec. 2013.

[16] P. Mach and Z. Becvar, "Mobile Edge Computing: A Survey on Architecture and Computation Offloading," *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[17] D. Wu, G. Shen, Z. Huang, Y. Cao, and T. Du, "A Trust-Aware Task Offloading Framework in Mobile Edge Computing," *IEEE Access*, vol. 7, pp. 150 105–150 119, 2019.

[18] S. Pan, Z. Zhang, Z. Zhang, and D. Zeng, "Dependency-Aware Computation Offloading in Mobile Edge Computing: A Reinforcement Learning Approach," *IEEE Access*, vol. 7, pp. 134 742–134 753, 2019.

[19] F. Samie, V. Tsoutsouras, L. Bauer, S. Xydis, D. Soudris, and J. Henkel, "Computation offloading and resource allocation for low-power IoT edge devices," in *3rd World Forum on Internet of Things*. IEEE, 2016, pp. 7–12.

[20] M. Chen and Y. Hao, "Task Offloading for Mobile Edge Computing in Software Defined Ultra-Dense Network," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 587–597, 2018.

[21] A. Shakarami, A. Shahidinejad, and M. Ghobaei-Arani, "A review on the computation offloading approaches in mobile edge computing: A game-theoretic perspective," *Software: Practice and Experience*, vol. 50, no. 9, pp. 1719–1759, 2020.

[22] J. Li, M. Dai, and Z. Su, "Energy-aware Task Offloading in Internet of Things," *IEEE Wireless Communications*, pp. 1–6, 2020.

[23] Zolertia, "Zolertia RE-Mote Revision B Internet of Things hardware development platform, for 2.4-GHz and 863-950MHz IEEE 802.15.4, 6LoWPAN and ZigBee® Applications," Barcelona, Spain, Datasheet ZOL-RM0x-B, Sep. 2016, v1.0.0.

[24] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[25] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *Journal of the ACM*, vol. 43, no. 4, pp. 685–722, Jul. 1996.

[26] F. C. Gärtner and S. Pleisch, "Failure Detection Sequencers: Necessary and Sufficient Information about Failures to Solve Predicate Detection," in *Distributed Computing*, D. Malkhi, Ed. Berlin, Heidelberg: Springer, 2002, pp. 280–294.

[27] A. Arora and S. S. Kulkarni, "Detectors and correctors: a theory of fault-tolerance components," in *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183)*, 1998, pp. 436–443.

[28] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985.

[29] M. Bradbury, A. Jhumka, and T. Watson, "Trust Assessment in 32 KiB of RAM: Multi-application Trust-based Task Offloading for Resource-constrained IoT Nodes," in *The 36th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC'21. Virtual Event, Republic of Korea: ACM, 22–26 March 2021, pp. 1–10.

[30] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," techreport, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[31] M. Bradbury, A. Jhumka, and T. Watson, "Dataset for: Trust Trackers for Computation Offloading in Edge-Based IoT Networks," Dec. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4339398