TinyEVM: Off-Chain Smart Contracts on Low-Power IoT Devices

TinyEVM: Off-Chain Smart Contracts on Low-Power IoT Devices

Christos Profentzas*, Magnus Almgren*, Olaf Landsiedel†*
*Chalmers University of Technology, Gothenburg, Sweden
†Kiel University, Kiel, Germany
chrpro@chalmers.se, magnus.almgren@chalmers.se, ol@informatik.uni-kiel.de

***Abstract*—[1]With the rise of the Internet of Things (IoT), billions of devices ranging from simple sensors to smart-phones will participate in billions of micropayments. However, current centralized solutions are unable to handle a massive number of micropayments from untrusted devices.**

Blockchains are promising technologies suitable for solving some of these challenges. Particularly, permissionless blockchains such as Ethereum and Bitcoin have drawn the attention of the research community. However, the increasingly large-scale deployments of blockchain reveal some of their scalability limitations. Prominent proposals to scale the payment system include off-chain protocols such as payment channels. However, the leading proposals assume powerful nodes with an always-on connection and frequent synchronization. These assumptions require in practice significant communication, memory, and computation capacity, whereas IoT devices face substantial constraints in these areas. Existing approaches also do not capture the logic and process of IoT, where applications need to process locally collected sensor data to allow for full use of IoT micro-payments.

In this paper, we present TinyEVM, a novel system to generate and execute off-chain smart contracts based on sensor data. TinyEVM's goal is to enable IoT devices to perform micro-payments and, at the same time, address the device constraints. We investigate the trade-offs of executing smart contracts on low-power IoT devices using TinyEVM. We test our system with 7,000 publicly verified smart contracts, where TinyEVM achieves to deploy 93% of them without any modification. Finally, we evaluate the execution of off-chain smart contracts in terms of run-time performance, energy, and memory requirements on IoT devices. Notably, we find that low-power devices can deploy a smart contract in 215 ms on average, and they can complete an off-chain payment in 584 ms on average.

***Keywords*-Internet of Things, Blockchain, Smart Contracts, Payment Channels, Off-chain, Ethereum**

I. Introduction

As the Internet of Things (IoT) becomes deeply integrated into our daily lives, new opportunities emerge. For example, cities nowadays embed sensors into parking lots to measure occupation. This capability, in turn, allows for new application scenarios, such as smart parking. Once a car approaches an empty parking lot, the lot can automatically inform the car about the hourly parking fees (based on location, time, or other locally set parameters), and engage in their payment when the car drives away. This scenario belongs to a much more generic application setting where the plethora of IoT devices are frequently interacting in two phases. First, they agree on the conditions related to an activity (e.g., the parking fees). Second, at a later time, they are executing/ensuring these conditions have been fulfilled (e.g., payment of the fees). A key challenge here is that the two IoT devices ordinarily do not trust each other, as they are, for example, owned or operated by different entities.

A potential solution to this challenge, worth exploring, are blockchains and their corresponding smart contracts [1], [2]. In principle, via a smart contract stored in a blockchain, a vehicle, and a parking sensor could agree on hourly parking fees, and at a later point in time, enforce the payment (e.g., through micropayments). However, blockchain technologies and smart contracts of today assume powerful nodes that can communicate and synchronize frequently. Ethereum [3], for example, uses a virtual machine to execute smart contracts, where clients need to connect to the blockchain both to upload their transactions and to query for updates.

Contrary, the nodes in IoT networks face constraints in energy, memory, and computation capabilities, making the requirements of current state-of-the-art blockchain technologies an ill fit for the IoT ecosystem. For example, today's resource-constrained devices have some tens of kilobyte memory, which is quickly exceeded by code and state information of smart contracts. High bandwidth, always-on connectivity with 4G or 5G, is infeasible in terms of energy consumption and hardware costs for many applications that shall operate for years on battery power. Moreover, cellular network coverage is far from ubiquitously available. Energy-efficient LPWAN technologies such as LoRa and SigFox, in turn, do not provide the required bandwidth for direct on-line transactions between a smart device and the cloud.

Furthermore, to allow IoT devices to play a central role in future micro-services (e.g., smart parking), they must be able to provide a local context (e.g., sensor data) for the conditions related to the activity about to take place. However, most smart contracts are not well designed to handle input from the outside world. While Oracles [4], [5], as a third-party information source, can supply verified data

from Internet-connected sources, there is no direct way for a smart contract to trigger a sensor reading and actuator setting on the IoT sensor-node. Overall, we recognize a gap between high-level blockchain architectures and the need for additional services and the capabilities of IoT devices.

To overcome these challenges, we design TinyEVM, a novel architecture to execute off-chain smart contracts on low-power IoT devices. We begin by revealing the design challenges of an application scenario for payment channels using off-chain smart contracts and introducing three novel approaches. Firstly, we design the on- and off-chain smart contracts considering the trade-offs between the device constraints and the off-chain protocol. We remove the need for active synchronization of payment channels by using a logical clock. Secondly, we customize the Ethereum Virtual Machine (EVM) to run on resource-constrained IoT devices with just a few kilobytes of memory. Thirdly, we extend the EVM by introducing specific IoT opcodes to allow smart contracts to directly interact with the sensors and actuators of the local IoT device. Our goal is to enable smart contracts written for EVMs to benefit from the large pool of existing contracts and established toolchains for the design, implementation, and verification of smart contracts.

To summarize, our contributions are as follows:

- We design and implement TinyEVM, an open-source[2] system to enable and scale (micro)payments on low-power IoT devices.
- We devise a virtual machine to execute off-chain smart contracts on resource-constrained devices.
- We introduce the concept of specific IoT-opcodes to unify the logic of interacting with sensors and actuators inside a smart contract.
- We quantify the performance of TinyEVM in terms of computation, delay, memory, and energy consumption. Notably, we find that low-power devices (TI-CC2538) can deploy a smart contract in 215 ms on average. The node can complete an off-chain payment in 584 ms on average.
- We finally provide a discussion of implementation challenges and trade-offs regarding off-chain protocols for resource-constrained devices.

Outline. We organize this paper with the following structure. In Section II, we provide the necessary background for TinyEVM. In Section III, we provide an overview of our application scenario and the design requirements. In Section IV, we provide the system design of TinyEVM. In Section V, we provide a security analysis of our design. We present the evaluation results in Section VI. Finally, we provide the related work in Section VII and the conclusion in Sections VIII.

[2]https://github.com/chrpro/TinyEVM

II. Background

In this section, we provide the essential background to understand the concepts of Blockchains, Smart Contracts, the Ethereum Virtual Machine, Payment Channels (PC), and the Plasma framework.

A. Overview of Blockchains

A blockchain is a distributed ledger replicated by multiple nodes and kept consistent via a consensus protocol. In cryptocurrencies like Bitcoin and Ethereum, the protocol is called mining. With mining, each node can create a new state by solving a probabilistic mathematical puzzle.

As blockchains became popular, their scalability and performance limitations became apparent [6], [7]. The research community responded with several proposals to scale the blockchains like sharding [8], [9], consensus algorithm variations [10], and trusted execution [5]. In this paper we focus on three prominent proposals: (a) side-chains [11], (b) payment channels [12], [13], and (c) payment networks [14], which we further described below.

B. Smart Contracts and the Ethereum Virtual Machine

Smart contracts are executable programs stored as bytecode in the blockchain. They highly extend the use of a blockchain as they can programmable change its state. For example, with the so-called Ethereum Virtual Machine (EVM), each node participating in the consensus of the blockchain executes the smart contracts on its local EVM and validate the correctness of the created new state.

The EVM is a Quasi-Turing complete machine [3] to execute state-transitions in the blockchain. The EVM is a 256-bit stack-based machine executing bytecode statements. Each statement consists of an opcode, with 71 active (discrete) opcodes at the time of writing. The machine avoids an infinite execution of the bytecode by charging a fee for each execution statement, the so-called gas. This fee inhibits micro-payments to be affordable, which is why payment channels have been suggested (see below). If a smart contract runs out of gas in the middle of execution, it is aborted.

The current design of EVM treats smart contacts as sequential programs with no support for concurrency. Moreover, EVM does not allow smart contracts to have access to data outside of the network, limiting their usefulness. For a smart contract to include sensor data, current solutions use services such as Oracles [4]. Oracles act as a third-party information source, and supply verified data from Internet sources. TinyEVM proposes a novel approach to include IoT opcodes inside the EVM, where the smart contract can have access to the sensors and actuators of the device.

C. Payment Channels

The fee(s) for each payment in the blockchain often makes repeated micropayments unaffordable. Prominent proposals

to overcome this limitation are off-chain protocols like payment channels (PC) [12], [13].

With PC, two parties can swiftly exchange small payments and postpone updating the blockchain (avoiding the fees) until they reach a final state. The parties pre-agree and sign a smart contract, which locks a specific amount of money for a specified period. Later, any participant can unlock the funds by providing a final state signed by both participants within the pre-agreed period. As such, the payment channel is a combination of three distinct concepts: 1) **multi-signature addresses**, 2) **time-locks**, and 3) **hash-locks**.

Multi-signature addresses require n-of-n signatures to unlock its funds. The typical case is a 2-of-2 signature address that requires the approval of both parties to unlock the fund. A **time-lock** restricts the validity of the multi-signature to a limited time. A **hash-lock** requires the revealing of the pre-image of a secret hash value to consider a payment as valid. The payment channel requires at least two on-chain messages to the public blockchain. One message to open the channel and lock the desired amount, including the hash-locks and time-locks. Depending on the design, the channel allows the owner to send messages to update the status or extend the lock-period. With an open channel, two parties perform off-chain payments by exchanging signatures. When they reach the time to close the channel, they reveal the secret-hash.

There are two main extensions of payment channels. First, payment networks [12], [14] reuse existing user channels to route payments off-chain. Second, state channels [15]–[17] provide a general use of channels to store state changes for any application. TinyEVM builds on the design concepts of payment channels and adapts them to the specific requirements of IoT applications.

D. Side-Chains

Another proposal to scale blockchains is the idea of side-chains [11], [18]. A typical side-chain system includes three main components: 1) an **on-chain smart contract**, 2) **side-chain(s)**, and 3) an **exit function**.

The on-chain smart contract is published in the main-chain, and it acts as a bridge between several side-chains. The smart contract locks the funds in the main-chain that the side-chains can circulate as off-chain tokens. The nodes participating in the side-chain are responsible for maintaining the side-chain. Each side-chain has a mechanism for validating blocks and a fraud-proof mechanism. The fraud-proof(s) is used by the users to report malicious users trying to exit on the main-chain. The exit function allows off-chain nodes to claim the tokens from the side-chain(s). Finally, any node can challenge an exit request on the main-chain using a fraud-proof.

III. TinyEVM Overview

This section describes the motivation behind TinyEVM. First, we introduce a simple application scenario and the parties involved therein. Second, we present the system requirements and challenges for low-power devices. Third, we motivate our threat model based on the application scenario.

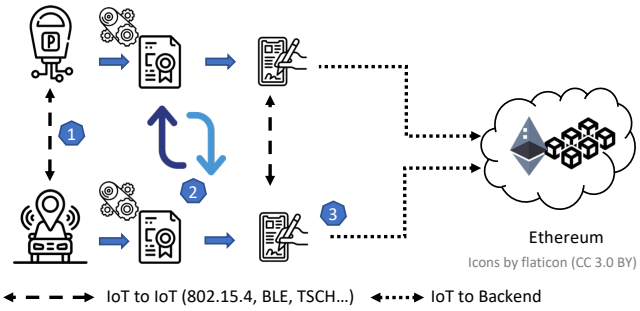


Figure 1: The parking application scenario: 1) The vehicle and the parking lot communicate via a short-range protocol. 2) They open an off-chain payments channel with an initial deposit and perform offline payments. 3) A node can at anytime submit a final state to the blockchain, in our case Ethereum.

A. Application Scenario: Smart Parking

Nowadays, smart parking lots equipped with sensors can detect occupation, and they can interact with a vehicle occupying a spot, for example, via low-power wireless technologies. We envision a marketplace where a car owner and a parking company negotiate the terms of parking and perform micropayments. For this, the car owner and the parking company publish a template smart-contract to a blockchain, which includes the necessary payment information.

When a vehicle approaches the parking lot, and the devices come within range, the lot can initiate the smart contract and create an off-chain payment channel with the vehicle via low-power wireless technologies, see Figure 1. The channel includes the common initial deposit of funds by the vehicle. During the parking, they may do multiple transactions and interactions, such as hourly payments or updates on the payment rates based on the time of day. At the end of the parking, they close the off-chain channel and sign the final state, which the parking lot can publish to the blockchain to claim the payment.

B. System Requirements

From the above scenario, we derive the following requirements for the application:

Sensor utilization: The parking lot would like to charge the vehicle owner based on the location of the parking spot, time of day, and possibly other locally relevant conditions, such as the parking availability. Thus, prices can vary and have to be agreed on, for example, via a smart contract.

Low latency: Both parties expect the whole process of negotiating and charging for the parking to be automated and

take place in the order of seconds. For this reason, we favor short-range wireless communication for our application scenario.

Low energy consumption: The parking service expects to depend on cheap devices with long battery life-time. The vehicle-owner expects a cheap device easily installed into a regular car.

C. System Challenges

Beyond the challenge of designing a off-chain protocol, we face other system challenges. To ensure compatibility with off-the-shelf smart contracts and to be able to benefit from the wide variety of tools for smart contract writing, testing, and verification designed by the Ethereum community, one key design goal of TinyEVM is to natively support the Ethereum Virtual Machine (EVM) bytecode. However, EVM is a 256-bit word-size virtual machine. This word-size leads to two key challenges: (1) resource inefficiency and (2) the complexity of executing 256-bit operations on a 32-bit machine.

First, from a resource perspective, the EVM does not utilize the memory efficiently. All operations are based on 256-bit variables and addresses. As a result, every VM opcode, even a simple addition, operates on 256-bit variables. The main reason for this EVM specification was to ease cryptographic operations like the Keecak256, which works on 256-bit digests. However, the vast majority of variables in a smart contract rarely reach values that require 256-bit storage. Similarly, as our evaluation shows, smart contracts do not reach a size in code or data that they would need 256-bit address space. The result is an inefficient memory use that could prohibit their execution of smart contracts on IoT devices.

The second challenge is the run-time overhead of the virtual machine. The embedded hardware does not directly support 256-bit operations. Instead, we have to emulate 256-bit operations using a 32-bit micro-controller by implementing custom libraries and, as a result, executing a single EVM opcode requires in the order of hundreds of MCU cycles. This inefficiency may further limit the potential of Ethereum smart contracts on resource-constrained IoT devices.

D. Threat Model

In our scenario, we assume mutually-distrusting, rational parties using a payment channel to exchange payments. The threat model includes two potential threats and focuses on the inability of nodes to revoke previous states of the payment channel. Notably, the node that receives the payment faces the threat of the inability to report a misbehaving peer before the contest period expires. On the other hand, the node sending the payment faces the threat of the inability to unlock her money from the channel.

The receiver expects a **non-repudiation** property of the system. After several payments, none of the parties will be able to deny the participation of the exchange. The parking sensor will be able to claim the money from the blockchain at any time for the time the car has stayed there. On the other hand, the sender expects a **finite** property of the payment channel, which means the channel eventually will close, and the sender can reclaim any remaining money when it leaves.

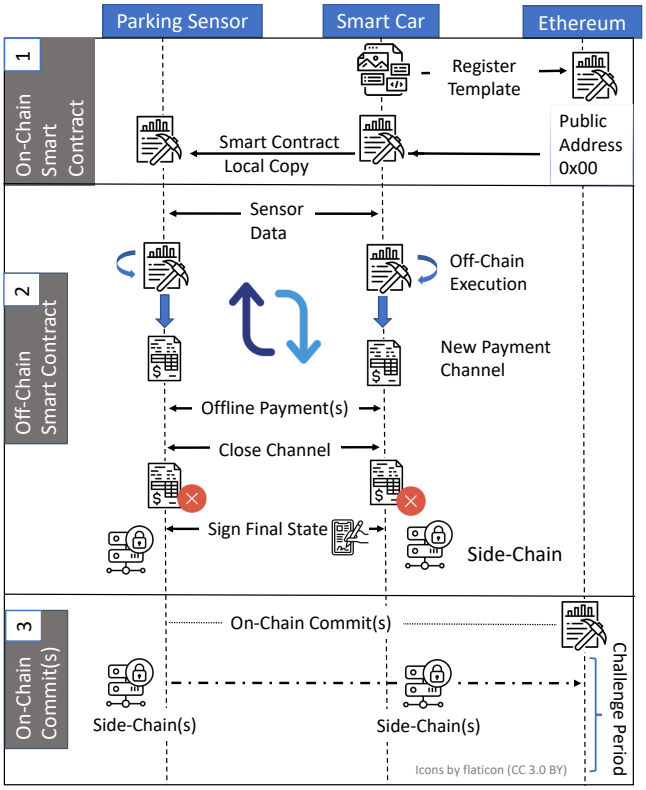


Figure 2: The system includes three phases: 1) Publishing the on-chain template smart contract. 2) Creating an off-chain payment channel and signing (multiple) payments. 3) On-chain commit of a final state, which activates the challenge period, and nodes can dispute with their local side-chains.

IV. TinyEVM System Design

TinyEVM makes four design contributions: First, it separates the transactions of IoT applications into three high-level phases: 1) The on-chain smart contract, 2) the off-chain smart contract, and 3) the on-chain commit(s). Second, TinyEVM customizes the Ethereum Virtual Machine (EVM) to address the resource constraints of IoT devices while staying compatible with the native EVM language. Third, we introduce a template design for smart contracts and the separation of on- and off-chain functionality. Fourth, we extend smart contracts and the EVM with IoT opcodes to interact with onboard IoT sensors and actuators. The new opcodes allow IoT devices to include sensor data and sensor actuation as a part of smart contract development.

Component	EVM	TinyEVM
Stack memory	256-bit	256-bit
Random access memory	8-bit	8-bit
Storage space	256-bit	8-bit
Operation opcodes	27	27
Smart contract opcodes	25	21
Memory opcodes	13	13
Blockchain opcodes	6	-
IoT opcodes	-	1

Table I: Comparison of the original EVM and the TinyEVM specifications. The word size of the stack and random access memory remains the same. TinyEVM removes the opcodes related to (on-)blockchain operation, it uses 8-bit side-chain storage space, and it introduces new IoT-specific opcodes.

A. On- and Off-Chain Transactions: Three Phases

The deployment and execution of smart contracts include three high-level phases, visible in Figure 2.

1) On-chain smart contract. A node publishes a smart-contract as a template in the blockchain, which includes the constructor for off-chain payment channels. The node makes a deposit to be charged for parking services, which works as an insurance in case of a dispute. This operation is similar to the on-chain contract used in side-chains. The on-chain smart contract serves as a bridge between the blockchain and our off-chain payment channels.

2) Off-chain smart contract. The nodes use the template to deploy a new off-chain payment channel using a unique monotonic counter (logical clock) as an identifier. The off-chain payment channels are locally generated smart contracts that allow the use of sensor data. Nodes may use the sensor readings, actuator interactions, or data received from other IoT devices as a part of the smart contract. The sensor data can range from weather conditions or device location, combined with the knowledge about the occupation of nearby parking spots – derived, for example, from the LIDAR data of a modern car – which can be used to evaluate and negotiate the parking fees. The nodes continue with the signing and exchanging of off-line payments until they close the payment channel. The nodes can open and close an arbitrary number of payment channels, but limited to the money deposited and locked in the on-chain smart contract.

3) On-chain commit. At any time, a node can exit from an off-chain channel by publishing a final channel state or a (side-chain) log of its local execution. Our commit function checks the logical clock of the channel and the validity of the signatures. In the case of a correct state, the new state is appended to the tree of the on-chain smart contract. The other node can challenge the state using the local log(s) of the off-chain payments. Finally, there is an exit function that a node can activate, which stops further updates from off-chain channels. The activation of the exit function starts the expiration period, and then it will dissolve the on-chain smart contract and return any unspent money. During that time, the other node can dispute the latest state and claim the insurance money, as common for established side-chains.

B. Customized Ethereum Virtual Machine

We enable smart contracts on IoT devices by customizing the Ethereum Virtual Machine (EVM) to meet the constraints of IoT devices, especially in terms of device memory. In Table I, we list the specifications of the original EVM compared with our customized one. There are three types of memory that a smart contract can use: 1) stack memory, 2) random-access memory, and 3) storage memory. We achieve to meet the memory requirements without the loss of functionality and compatibility of the IoT device.

The original EVM defines a 256-bit word machine, which is not directly supported on a 16-bit or 32-bit micro-controller. However, we keep the same word-size for compatibility reasons, and emulate a 256-bit word-size in our implementation. This implementation allows us to use the original Ethereum bytecode with no modifications. However, the storage space is irrelevant for off-chain executions. These operations are needed only for the main-chain. For the off-chain computations, we utilize an 8-bit storage space to store only the side-chain created by the IoT nodes.

We list the machine opcodes into five categories. First, the operation opcodes define the necessary computations, like addition and multiplication. The original EVM supports 27 operations, and TinyEVM supports all of them. Second, the smart contract opcodes are related to smart contract execution like method calls, and returns. TinyEVM supports the necessary operations except for the GAS operations. There is no charging for the off-chain computations as all operations are executed locally. Third, the memory opcodes are related to operations on memory like store and load, and TinyEVM supports all of them. Fourth, the blockchain opcodes are used to get information from the blocks of the blockchain. TinyEVM does not support any block-related opcode since there is no access to the blockchain during local execution. Finally, TinyEVM introduces a novel opcode for IoT sensor data. This extension allows us to include sensor data inside the smart contract.

We observed that the original EVM includes unused opcode(s) that are not currently in use. Thus, we utilized one of the unused opcodes to introduce the IoT sensor functionality. In detail, we use the 0x0c undefined opcode to represent the action of sensing or actuating on the device. Details, such as which sensor to use and additional parameters are given as options to the opcode. This allows us to include arbitrary types of sensor data, and the TinyEVM hides the implementation details from the user.

C. On-Chain Smart Contract

For our purposes, we assume that the entity providing a service (e.g., the parking service) has published the parking

```
contract Template {
  address[] PaymentChannels;
  uint Balance;
  address payable public Receiver;
  uint Logical-Clock = 0;
  MerkleSumTree Side-Chain-Root;

  function CreatePaymentChannel (uint64 Money)
      public  {
      newPaymentChannel = new PaymentChannel(
          receiver, Money);
      PaymentChannels.push(newPaymentChannel);
      Logical-Clock += 1;
  }
  function OnChainCommit{...} //user specific
  function Challenge{...} //user specific
}
```

Listing 1 : Factory Template in Solidity

```
contract PaymentChannel {
 address payable public Sender;
 address payable public Receiver;
 uint public sensor_data;

constructor(address payable _recipient) public
    payable {
  sender = msg.sender;
  recipient = _recipient;
  assembly{
      0x0c //IoT sensor opcode
      sstore(0x0c) // Store sensor data
  }
}
function close(uint amount, bytes memory
    signature) public payable {
  require(msg.sender == recipient);
  require(isValidSignature(amount, signature));
  recipient.transfer(amount);
  selfdestruct(sender);
}
}
```

Listing 2 : Payment Channel in Solidity

conditions as a smart contract template on the blockchain. The template includes all rules that the two parties need to create and use an off-chain payment channel. Upon accepting the conditions, the user (e.g., the owner of the car) locks the desired amount to be used for the services. Alternatively, if both parties have to negotiate some details, an additional negotiation phase is possible to construct the template [19].

The template is a factory-smart-contract [20], which can create and deploy child contracts dynamically, see Listing 1. The child contracts in our scenario are the payment channels, see Listing 2.

D. Off-Chain Smart Contract

The second phase in Figure 2 starts when both entities come within range of their low-power wireless technologies, e.g., a smart-car and smart-parking lot come within range. The nodes exchange their sensor data and transactions via a short-range protocol like TSCH [21] or BLE [22]. Please note that the design of TinyEVM is agnostic to the specific technology used. Both entities execute the bytecode of the template to generate an off-chain payment channel. The payment channel includes an ID and the sequence number, which uniquely identifies each transaction on the channel. The sequence number acts as a logical clock, which is different from the real-time bound of the original concept of payment channels. By using sequence numbers, we can determine the causal order of the payments. With the casual order, the channels can capture the order in which the payments happens, but not the actual time that they occur. The use of logical clocks loosens the requirements for communication and synchronization.

Each device maintains a sequence number that uniquely identifies each of its transactions by simply incrementing a counter for each new transaction. The sequence number is later used for verification and ensures that no device skips reporting any transactions. With the off-chain payment channel, the two nodes can perform several off-chain payments by exchanging signed transactions (using their low-power wireless radios). The signed off-chain payments are stand-alone artifacts that can claim money from the main-chain. The signed payment includes information on the payment channel ID and its unique counter, which makes it trivial to verify the logical order. A node can report either the payment or the final state of the channel, which aggregates all other previous payments. Each execution of the payment channel extends the local (side-chain) log of the node, which links each state with the previous. The local (side-chain) log uses the root published on the main-chain smart contract, which allows verification of the logical order of the executions and ensures that no transactions are omitted. The nodes use the off-chain payment channel repetitively to perform several payments until they close it. The total amount of payments is limited by the funds locked in the main-chain.

E. On-Chain Commit & Challenge Period

At any time, a node can submit a signed final state of a closed off-chain payment channel. The sequence number of the channel allows the nodes to retrieve the money asynchronously. The node can submit a state that happened after the latest submit without providing the details of the actual time that it happened. Every time the on-chain contract receives a request, it verifies the signed states and updates its sequence number to the highest that received.

The on-chain smart contract uses a Merkle-Sum-Tree [11], which has the sum of the payments and the hash value. The sum value is used as a validation condition along with the hash value. This condition makes it possible for auditing the sum of the payments. Each payment adds to the overall sum, and if it exceeds the allowed range, the payment is invalid,

and the other node can claim the insurance money. In our system, we further extend the validation condition with the sequence numbers. Reporting a state with a higher sequence number accumulates the changes of the previous states.

Finally, the sensor node can activate the exit function by submitting a signed final state. This action restricts further submission and starts the challenge period. The other node can submit a transaction with a higher sequence number value to claim the insurance money. As each transaction is signed, it is not disputable.

V. Security Analysis

Similar to other off-chain systems [11], [23], TinyEVM does not entirely prevent double-spending fraud but instead makes it unprofitable. We design our system based on three security properties. 1) We can detect any fraud using the sequence numbers and the signatures of the participants. 2) We introduce proper punishment and incentives for the participants to report any misbehavior. 3) We have a time-limit in which a party can claim the money deposited in the on-chain contract.

Detection: Each time a node performs a transaction or closes a channel, it increases the sequence number, and it eventually will report the local state to the blockchain. The on-chain smart contract always stores the most recent state, i.e., the one with the highest sequence number. As a result, the sequence number prevents a node from misbehaving by reporting old states. Reporting a signed transaction or state with a higher sequence number denotes a valid next state.

A node could exchange a transaction with another node, and it may skip to report or even try to delete the transaction. If the other node behaves correctly, it will upload all the transactions, and the on-chain smart contract detects the misbehaving node. Thus, a transaction will only go unnoticed if both involved parties are misbehaving. We argue that it is very uncommon for two parties to conspire in this way, as they commonly do not share the same goal.

Non-repudiation: One party could claim the money from the blockchain at any time by providing a signed state. On the other hand, the other party can close the channel to refund any unspent money at any time. The system is based on incentives that penalize misbehavior. The first party has the incentive not to overspend the locked funds; otherwise, the insurance money will be lost. The second party has the incentive to report the last payment before the template expires.

Time-limit: The template has an exit function that allows the sensor owner to start the process to renounce any unspent money. This time-limit is in order of days similar to the popular framework (e.g., Plasma, which has a seven-day bound) [11].

VI. Performance Evaluation

In this section, we present the evaluation of TinyEVM on low-power IoT devices. The evaluation responds to the following questions. (a) Is TinyEVM technically feasible on low-power IoT devices? (b) What is the performance of executing a smart contract on a low-power device? (c) What is the overhead in terms of computation, memory, and energy consumption of the off-chain functionality, including both wireless communication and local processing.

Outline. First, we list the implementation details and target platforms. Second, we present the evaluation of the customized Ethereum Virtual Machine (EVM) on low-power devices. This part of the evaluation represents a macro-benchmark of the system regarding the ability to deploy smart contracts on low-power devices. Third, we present the evaluation of the off-chain functionality in terms of run-time performance, energy, and memory requirements for both communication and local computation. The off-chain evaluation represents a micro-benchmark of the system, and it provides details on executing the off-chain payment-channel application.

A. Experimental Setup

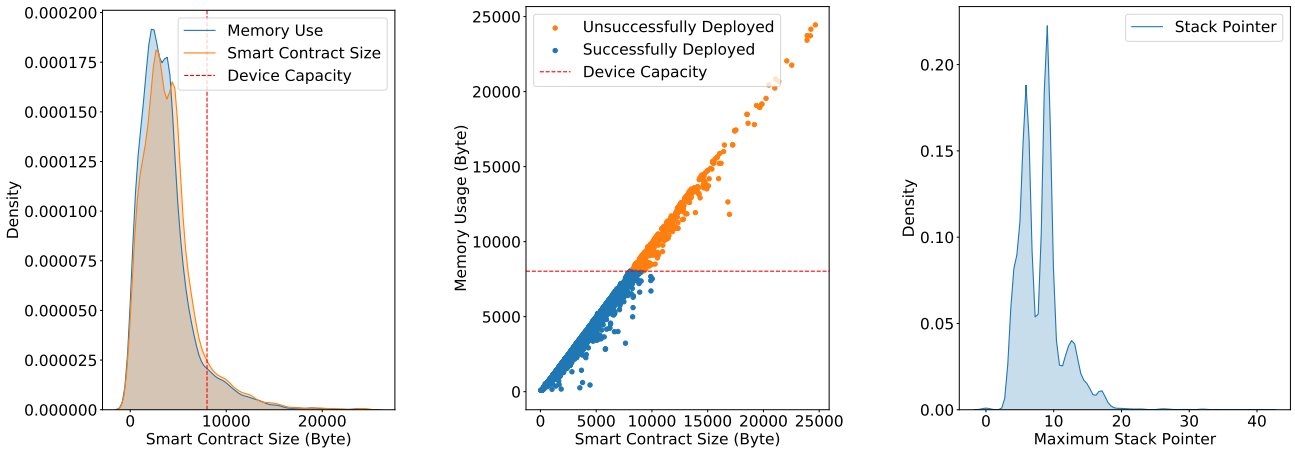
Implementation. We implement the Ethereum Virtual Machine (EVM) in C as a module for the Contiki-NG OS. For wireless communication, we use the TSCH protocol stack provided by Contiki-NG. We support smart contract deployment up to 8 KB of bytecode. We implement EVM as a 256-bit word size machine with 3 KB of stack, 8 KB of random access memory, and 1 KB for off-chain storage. A comparison between the original specifications and TinyEVM is presented in Table I.

Hardware Setup. Building on cryptographic primitives, our implementation targets sensor nodes with cryptographic hardware support. We use Openmote B that is based on the TI-CC2538 SoC [24]. The SoC runs a 32-bit ARM Cortex-M3 CPU at 32 MHz, 32 KB of RAM, and 512 KB of ROM. Moreover, the SoC features a cryptographic engine at 250 MHz and an 802.15.4 radio transceiver. Please note that TinyEVM is not bound to this particular platform, and it can be deployed on any platform supported by Contiki-NG, as long as it has a cryptographic co-processor, sufficient resources, and a 802.15.4 radio interface.

B. Ethereum Virtual Machine on IoT Devices

We evaluate the resource efficiency of TinyEVM and its ability to deploy off-the-shelf smart contracts. For this, we collect roughly 7,000 publicly available smart contracts to test our platform. The smart contracts are verified by Etherscan.io, a widely used blockchain explorer.

1) Memory Requirements: The deployment of a smart contract starts with the initialization of the smart contract using its constructor function. This function initializes all the variables, and it will take the initial steps to make the smart contract executable. Finally, it will return the actual bytecode that will be installed on the device.



(a) The distribution of the memory requirements for 7,000 smart contracts. We are able to deploy 93% (5,953) of the smart contracts on the low-power device. The memory capacity is set to 8 KB. For the remaining 7% we would need more memory on the device.

(b) Device memory usage in relation to the smart contract size. There is a positive correlation between the smart contract size the device memory requirements. The memory required for the deployment is never longer than the size of the contract.

(c) The use of the stack memory of the successful deployed smart contracts. The figure shows the maximum value reached by the stack pointer.

Figure 3: The graphs presenting the memory usage of the smart contract deployment. This includes the density distribution regarding the memory and stack usage of the virtual machine.

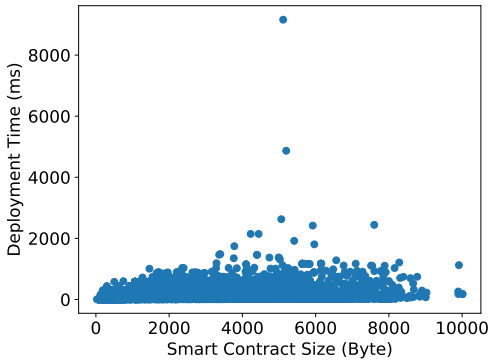


Figure 4: The time of deploying a smart contract in relation to its bytecode size. The average time is 215 ms, but we can notice there are some outliers.

In Figure 3a, we see the distribution of the smart contract size and the memory usage in TinyEVM after the deployment. We observe that the average size of a smart contract is 4 KB (see also Table II). The maximum size is 25 KB and the minimum size is 28 Bytes. The sizes of the smart contracts are within the capacity of the device that facilitates 32 KB of RAM. The deployment limit is set to 8 KB (red-dotted line); If we want to deploy larger smart contracts, we need to allocate less memory for the system configuration (see Table III, e.g., stack size). Such allocation will lead to execution failures, e.g., stack overflows, and we argue that 8 KB represents a favourable memory allocation point.

In our experiment, we successfully deploy 93% (5,953) of the public smart contracts on the low-power device without any modification. All other contracts fail due to resource limitations. In Figure 3b, we observe the memory usage needed to deploy the smart contracts. We notice the positive correlation between memory use and the size of the smart contract. However, the final deployment never requires more memory than the actual size of the smart contract. This behavior allows the device to deploy some outliers with bytecode size higher than 8 KB, but with a final deployment requirement of less than 8 KB.

We continue our analysis with the virtual machine's stack usage during the experiments. In Figure 3c, we present the distribution of the maximum Stack Pointer (SP) during execution. We can observe the majority of the smart contracts use a maximum of ten elements. In Table II, we observe that the maximum SP is 41 elements, with an average of 8 elements. As a reminder, the Ethereum specifies a maximum SP of 1024 elements. From our experiment, we see a tendency of developers to write small, concise smart contracts, which rarely need more stack during execution. However, the execution of deployed smart-contract functions can vary depending on the parameters. The evaluation of these functions is hard to define in practice as their parameters are not known before the actual execution. However, our evaluation gives some insights regarding the design choices for the virtual machine. Overall, we conclude that in terms of random access memory, stack memory, and storage memory of TinyEVM, we support the majority of the 7,000 publicly available smart contracts.

2) Deployment Execution Time: In Figure 4, we present the deployment time (in ms) compared to the size of the bytecode. As a first observation, there is no correlation between the size of the bytecode and the deployment time. However, we only perform the deployment of the smart contract in our evaluation with the default parameters.

We see in Table II that the average execution time is 215 ms, with a standard deviation of 277. It is worth to notice that we observe some outliers that need more time to deploy the smart contract, which highly depends on the nature of the opcodes they use. The maximum time we observe is 9.2 seconds, showing that smart contracts can be deployed within seconds, even on resource-constrained IoT devices.

C. Off-chain Payment Channels

Next, we give insights into the memory, CPU performance, and energy consumption of the execution of off-chain payment channels on low-power devices. We run our experiments over 200 times, and we report the standard deviation when it is not negligible (as σ). For the energy consumption of Contiki-NG, we rely on the internal Energest module [25] that has a 30-microsecond resolution timer.

1) Memory Requirements.: We present the memory footprint of the payment channel in Table III and divide it into three main parts: (a) The Contiki-NG is the operating system including the necessary network stack and libraries, (b) the smart contract template to generate payment channels, and (c) the TinyEVM is our customized Ethereum Virtual Machine (EVM).

Contiki-NG itself consumes 33% of the available RAM. This module is necessary for the general functionality of the device and also provides the wireless protocol stack. The EVM has a significant impact and consumes 42% of the RAM. The smart contract template, which we implemented for this evaluation scenario, is deployed as bytecode and consumes only 5% of the RAM. Finally, the whole program consumes only 11% of the ROM. The deployed smart contract fully supports our smart parking application scenario, and the results underline that we have resources for significantly more complex application scenarios.

2) Cryptographic modules: Next, we evaluate the performance of the cryptographic functions. The keccak256 hash function is based on software implementation, as the cryptographic hardware of our platform does not support it. All the other cryptographic operations are performed by the cryptographic hardware running at 250 MHz. In Table V, we present the performance of each separate task. The average time to complete all cryptographic functions of a complete transaction round is 356 ms. The most time-consuming operation is the ECDSA signature, which takes 350 ms. We argue that this overhead, while significant, is feasible for a large number of applications. For example, in the car parking application scenario, it means a user will need less than a second to sign and exchange a valid transaction.

3) Energy consumption: We present the energy consumption of off-chain transactions on resource-constrained IoT devices. Focusing on the energy consumption of the off-chain payment channel, we report our results after the TSCH node discovery. Node discovery happens quickly [21], and the energy consumption is insignificant. Moreover, this discovery is specific to the TSCH protocol and would have a different footprint on other communication technologies such as BLE. In Figure 5, we depict the flow of the electric current (in mA) for a full-round of the off-chain process. The process involves three discrete pieces: wireless communication, the virtual machine execution, and the cryptographic engine.

As a first step, in the parking scenario, the nodes exchange their data. Our evaluation includes reading sensor data, in this case from the temperature sensor, to evaluate the overhead of IoT sensor and actuator operations. The smart car starts by sending its sensor data at 0.25 s, followed by the receiving of parking sensor data visible in Figure 5. Second, the car at 0.45 s executes the smart contract to create the off-chain payment channel. This execution takes on average 0.20 s with a σ of 0.1.

Third, the car signs a payment for the parking sensor, where the signature takes 0.35 s on average. In practice, such payment would be conducted at an application-specific rate, commonly in the order of minutes. For brevity, we include only one payment here. At the end of the parking, the car executes the off-chain payment channel to register the payment on the side-chain. This process takes on average 0.08 s with a σ of 0.01. Finally, the car exchanges signatures with the parking sensor.

In Table IV, we report the total energy consumption (in mJ) of the off-chain process. We notice that the major energy consumption (65%) comes from the cryptographic engine with 19.1 mJ. The wireless communication using the TSCH protocol contributes to 3,7 mJ (13%). Finally, the execution of the virtual machine contributes to a CPU consumption of 4,1 mJ (14%). In total, the off-chain process consumes 29,6 mJ. We observe that the cryptographic engine is the main energy consumer, while both the virtual machine and wireless communication consume considerably less.

By default, the OpenMote platform is powered by two standard AA alkaline battery of 2500 mAh. Thus, we can expect 10,000 Joules of energy from the cells, which allows us to perform roughly 333,000 payments. If we assume one payment on average every 10 minutes, this would lead to a battery life-time of more than six years. While this is merely an estimate and other factors such as energy consumption during deep sleep and battery leakage need to be considered, we argue that this order of magnitude of payments is practical for a wide range of application scenarios, including our parking example.

Measurement	Contract Size	Stack Pointer	Stack (Bytes)	Memory (Bytes)	Deployment Time (ms)
Max	10,058	41	3,056	8,056	9,159
Min	28	3	768	96	5
Mean	4,023	8	2,048	3,676	215
Std	2899	3	827	2,801	277

Table II: An overview of memory and deployment time of the 5,953 successfully deployed smart contract.

Component	RAM Bytes	RAM Percent	ROM Bytes	ROM Percent
Contiki-NG OS	10,394	33%	40,527	10%
TinyEVM	13,286	42%	1,937	1%
Smart Contract Template	2,035	5%	-	-
Total footprint	25,715	80%	53,239	11%
Available memory	6,285	20%	458,761	89%

Table III: Memory Footprint of the TinyEVM (max sizes) on CC2538, which facilitates 32KB of RAM and 512 KB of ROM.

State	Time [ms]	Current [mA]	Energy [mJ]
Cryptographic Engine	350	26	19.1
TX	32	24	1.6
RX	52	20	2.1
CPU @ 32 MHz	150	13	4.1
CPU @ LPM2	982	1.3	2.7
Total	1,566	-	29.6

Table IV: Derived energy consumption of running the contract signing protocol on the CC2538 SoC given a supply voltage of 2.1 V. Note we configure Contiki-NG to use the low-power mode 2 (LPM2) [24], when not active.

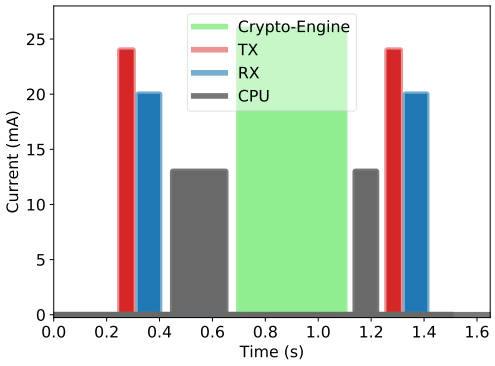


Figure 5: The electric current (in mA) drawn by a complete round of the off-chain payments channel. There are three discrete pieces involved: the wireless communication, the virtual machine execution, and the cryptographic engine.

VII. Related Work

We list the related work in five parts: (1) blockchains and IoT, (2) scaling of blockchains using off-chain protocols including payment channels and networks, (3) payment hubs, (4) side-chains and oracles, and (5) virtual machines in the context of IoT and wireless sensor networks.

Operation type	Mode	Time
ECDSA - Signature	HW	350 ms
SHA256 - Hash function	HW	1 ms
Keccak256 - Hash function	SW	5 ms
Total time		356 ms

Table V: Performance of cryptographic operations. Keccak256 is implement on software, the other operations are using the CC2538 cryptographic engine running at 250 MHz.

Blockchain and IoT. Previous proposals highlight the benefits of using a blockchain for IoT applications. IoTLogBlock [2] demonstrates the feasibility of creating and validating transactions using low-power devices for cloud-based blockchains like Hyperledger. However, IoTLogBlock is not applicable to public blockchains such as Ethereum and Bitcoin. AGasP [1] analyses the benefits of using smart contracts for IoT applications. This architecture assumes powerful nodes able to interact and synchronize with the main blockchain. TinyEVM proposes the off-chain execution of smart contracts and involves sensor data as part of the execution.

Payment Channels & Networks. Researchers have proposed several improvements and extensions to Payment Channels (PC). A major extension to PC is the payment networks, where users can reuse existing PCs to form a routing network. On the commercial side, the Lighting Network [26] was one of the first implementations of such networks for Bitcoin. The equivalent network for Ethereum is Raiden [27].

There are three main challenges to make PCs usable in practice. One challenge is to open a PC in both directions. Duplex micropayments [28] are an extension to allow the user to have this type of PC. Second, a user cannot reallocate the locked money in the channel. Revive [12] allows a user to rebalance the payment channels and to reallocate money to a channel without the cost of closing and reopening it. Third, there are privacy concerns regarding the ability of track payments. Bold [13] tackles privacy issues and ensures that multiple payments are unlinkable with the assumption that the participants use anonymized capital. Other proposals solve the privacy issues with different trade-offs, for example SilentWhispers [29], and SpeedyMurmurs [30]. However,

the above approaches assume active communication and synchronization among nodes, which is not a valid assumption in the context of IoT, especially resource-constrained IoT.

Payment Hubs. The idea of a payment hub is to use the nodes that have multiple open channels (hub) to circulate the payments. The other nodes need to connect to a hub node. There are three challenges for the payment hubs.

First, there are concerns about the anonymity of the payments. Tumblebit [17] makes the payments unlinkable by using an untrusted intermediary. Second, there is the involvement of the intermediary for each payment. This intermediary leads to performance issues. Perun [31] proposes the virtual payment hub to avoid this problem. Third, payment hubs can lead to collateral fragmentation. NOCUST [32] proposes the separation of the functionality of the payment hub to two components. First, an off-chain operator server handles every transfer. Second, an on-chain smart contract verifies the payments. Finally, Ye et al. [14] propose a system (Boros) to shorten the payment path for the hub network. In the context of IoT, a payment hub allows us to scale the payment system, but several trade-offs need to be evaluated. This is one focus of our future research.

Side-Chains. This proposal [23] includes an on-chain smart contract acting as a bridge between several lighter and faster side-chains. The nodes can exchange the off-chain tokens that have a correspondence in the main blockchain.

The Plasma [11] framework is the proposal of the Ethereum team to scale the blockchain network. However, the current side-chains do not take into consideration the challenges of low-power IoT devices. Our system is built on top of these ideas, and we further extend the functionality of off-chain smart contracts to have access to sensors and actuators of the IoT device.

Oracles. An oracle provides a solution to the design inability of Ethereum to include data from the physical world (e.g., sensor data). TownCrier [5] provides a bridge between HTTPS-enabled data websites and the Ethereum blockchain. Moudoud et al. [4] show a test case of an IoT supply chain scenario using a network of oracles and smart contracts. Our system differs from these proposals since TinyEVM proposes a novel approach where the smart contract can have access directly to the sensors and actuators of the IoT device.

Virtual Machine for IoT. Several virtual-machines [33]–[35] have been proposed for IoT devices before to provide support for high-level languages such as Java. However, most of them define word-size from 8-bit to 32-bit indexes, which are natively supported. In TinyEVM, we design a 256-bit machine tailored for off-chain smart contracts, which brings new challenges. A limitation of the Ethereum Virtual Machine (EVM) is its limited support for concurrency, which TinyEVM inherits. One suggested solution for concurrent execution is to use speculatively parallel executions of smart contracts [36], however, these solutions are not designed for resource-constrained devices.

VIII. Conclusion

In this paper, we present TinyEVM, a novel system to perform off-chain payments using low-power IoT devices. TinyEVM allows deploying smart contracts from powerful nodes on a resource-constrained device. We also extend the functionality of the smart contracts to have access to sensor reading and actuation of the device as part of the high-level code, allowing the integration of could-services with IoT-nodes.

TinyEVM achieves a sweet spot between the scalability requirements of the blockchain and the off-chain computation. The design of TinyEVM focuses on the energy and memory requirements of low-power devices. Our evaluation shows the technical feasibility of executing off-chain smart contracts on IoT devices. We deploy 5,953 smart contracts with an average of 4 KB size with the average deployment time of 215 ms. Finally, we evaluate the execution of off-chain smart contracts in terms of run-time performance, energy, and memory requirements on IoT devices. Notably, we find that low-power devices can deploy a smart contract in 215 ms on average. The IoT node can complete an off-chain payment in 584 ms on average.

As future work, we will investigate the feasibility of payment networks and payment routing algorithms on low-power IoT devices. Also, we will improve some of the privacy concerns of off-chain payments.

IX. Acknowledgments

This work was supported by the Swedish Research Council (VR) through the project "AgreeOnIT", the Swedish Civil Contingencies Agency (MSB) through the projects "RICS" and "RIOT", and the Vinnova-funded project "KIDSAM".

References

[1] Y. Hanada, L. Hsiao, and P. Levis, "Smart Contracts for Machine-to-Machine Communication: Possibilities and Limitations," *IEEE Conference on Internet of Things and Intelligence System (IOTAIS)*, 2018.

[2] C. Profentzas, M. Almgren, and O. Landsiedel, "IoTLogBlock: Recording Off-line Transactions of Low-Power IoT Devices Using a Blockchain," in *IEEE Conference on Local Computer Networks (LCN)*, 2019.

[3] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger EIP-150," *Ethereum Yellow Papers*, 2017.

[4] H. Moudoud, S. Cherkaoui, and L. Khoukhi, "An IoT Blockchain Architecture Using Oracles and Smart Contracts: the Use-Case of a Food Supply Chain," in *IEEE Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2019.

[5] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town Crier: An Authenticated Data Feed for Smart Contracts," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[6] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[7] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains," in *ACM Conference on Management of Data (SIGMOD)*, 2017.

[8] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding," *IEEE Symposium on Security and Privacy (SP)*, 2018.

[9] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[10] L. M. Bach, B. Mihaljevic, and M. Zagar, "Comparative analysis of blockchain consensus algorithms," in *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018.

[11] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *Plasma.io*, 2017.

[12] R. Khalil and A. Gervais, "Revive: Rebalancing Off-Blockchain Payment Networks," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[13] M. Green and I. Miers, "Bolt: Anonymous Payment Channels for Decentralized Currencies," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[14] Y. Ye, J. Zhang, W. Wu, X. Luo, and J. Cao, "Boros: Secure Cross-Channel Transfers via Channel Hub," *arXiv*, 2019.

[15] S. Dziembowski, S. Faust, and K. Hostáková, "General State Channel Networks," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[16] A. Miller, I. Bentov, R. Kumaresan, C. Cordi, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," *arXiv*, 2017.

[17] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[18] F. Gai, C. Grajales, J. Niu, M. M. Jalalzai, and C. Feng, "Cumulus: A BFT-based Sidechain Protocol for Off-chain Scaling," *arXiv*, 2019.

[19] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart Contract Templates: essential requirements and design options," *arXiv*, 2016.

[20] P. Zhang, J. White, D. C. Schmidt, and G. Lenz, "Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps," *arXiv*, 2017.

[21] S. Duquennoy, A. Elsts, B. A. Nahas, and G. Oikonomo, "TSCH and 6tisch for Contiki: Challenges, Design and Evaluation," in *IEEE Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2017.

[22] B. A. Nahas, S. Duquennoy, and O. Landsiedel, "Concurrent Transmissions for Multi-Hop Bluetooth 5," in *IEEE Conference on Embedded Wireless Systems and Networks (EWSN)*, 2019.

[23] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "SoK: Off The Chain Transactions," *IACR Cryptology ePrint Archive*, 2019.

[24] Texas Instruments, "CC2538 system on chip for 2.4-GHz IEEE 802.15.4," www.ti.com.cn/cn/lit/ug/swru319c/swru319c.pdf, 2013.

[25] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He, "Software-Based on-Line Energy Estimation for Sensor Nodes," in *Proceedings of the 4th Workshop on Embedded Networked Sensors (EmNets)*. ACM, 2007.

[26] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," *https://lightning.network/*, 2016.

[27] R. Network, "What is the raiden network?" *https://raiden.network/*, 2018.

[28] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Symposium on Self-Stabilizing Systems*. Springer, 2015.

[29] G. Malavolta, P. Moreno-Sanchez, A. Kate, and M. Maffei, "SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks," *Network and Distributed System Security Symposium (NDSS)*, 2017.

[30] S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg, "Settling payments fast and private: Efficient decentralized routing for path-based transactions," *Network and Distributed System Security Symposium (NDSS)*, 2017.

[31] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "PERUN: Virtual Payment Channels over Cryptographic Currencies," *IACR Cryptology ePrint Archive*, 2017.

[32] R. Khalil and A. Gervais, "NOCUST-A Non-Custodial 2nd-Layer Financial Intermediary," *IACR Cryptology ePrint Archive*, 2018.

[33] N. Reijers and C.-S. Shih, "CapeVM: A Safe and Fast Virtual Machine for Resource-Constrained Internet-of-Things Devices," in *ACM Conference on Embedded Networked Sensor Systems (SenSyS)*, 2018.

[34] P. Levis and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks," in *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[35] R. Müller, G. Alonso, and D. Kossmann, "A Virtual Machine for Sensor Networks," in *ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[36] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding Concurrency to Smart Contracts," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 2017.