Hephaestus: Modelling, Analysis, and Performance Evaluation of Cross-Chain Transactions

Rafael Belchior [1], Peter Somogyvari [2], Jonas Pfannschmid [2], André Vasconcelos [2], and Miguel Correia [2]

[1]Blockdaemon; INESC-ID
[2]Affiliation not available

October 30, 2023

Abstract

Ecosystems of multiple blockchains are now a reality. Multi-chain applications and protocols are perceived as necessary to enable scalability, privacy, and composability. Despite being a promising emerging research area, we recently have witnessed many attacks that have caused billions of dollars in losses. Attacks against bridges that connect chains are at the top of such attacks in terms of monetary cost, and no apparent solution seems to emerge from the ongoing chaos.

In this paper, we present our contribution to minimizing bridge attacks. In particular, we explore the concepts of *cross-chain transaction, cross-chain logic,* and the *cross-chain state* as the enablers of the cross-chain model. We propose **Hephaestus**, the first cross-chain model generator that captures the operational complexity of cross-chain applications. **Hephaestus** can generate cross-chain models from local transactions on different ledgers realizing arbitrary use cases and allowing operators to monitor their cross-chain applications. Monitoring helps identify outliers and malicious behavior, which can help programmatically to stop bridge hacks and other attacks. We conduct a detailed evaluation of our system, where we implement a cross-chain bridge use case. Our experimental results show that **Hephaestus** can process 600 cross-chain transactions in less than 5.5 seconds in an environment with two blockchains and requires sublinear storage.

Hephaestus: Modelling, Analysis, and Performance Evaluation of Cross-Chain Transactions

Rafael Belchior*[†], Peter Somogyvari [‡], Jonas Pfannschmidt [†], André Vasconcelos*, Miguel Correia*
*INESC-ID, Instituto Superior Técnico, Universidade de Lisboa [†]Blockdaemon Ltd. [‡]Accenture

***Abstract*—Ecosystems of multiple blockchains are now a reality. Multi-chain applications and protocols are perceived as necessary to enable scalability, privacy, and composability. Despite being a promising emerging research area, we recently have witnessed many attacks that have caused billions of dollars in losses. Attacks against bridges that connect chains are at the top of such attacks in terms of monetary cost, and no apparent solution seems to emerge from the ongoing chaos.**

In this paper, we present our contribution to minimizing bridge attacks. In particular, we explore the concepts of *cross-chain transaction*, *cross-chain logic*, and the *cross-chain state* as the enablers of the cross-chain model. We propose `Hephaestus`, the first cross-chain model generator that captures the operational complexity of cross-chain applications. `Hephaestus` can generate cross-chain models from local transactions on different ledgers realizing arbitrary use cases and allowing operators to monitor their cross-chain applications. Monitoring helps identify outliers and malicious behavior, which can help programmatically to stop bridge hacks and other attacks. We conduct a detailed evaluation of our system, where we implement a cross-chain bridge use case. Our experimental results show that `Hephaestus`can process 600 cross-chain transactions in less than 5.5 seconds in an environment with two blockchains and requires sublinear storage.

I. Introduction

Recently, many initiatives and projects have appeared around the concept of blockchain interoperability (BI), where a multi-chain ecosystem is perceived as the enabler for a scalable and adaptable platform for various use cases [1]–[4]. To enable such an ecosystem, bespoke distributed ledger technology (DLT) interoperability solutions, such as cross-chain bridges (or simply bridges), are used to connect heterogeneous DLTs, i.e., DLTs with different privacy, security, decentralization, and scalability properties. Bridges are one of the most widely used classes of cross-chain applications. The total value locked in bridges peaked in March 2022 at over $25 billion dollars worth of assets [5], effectively reflecting the synergistic effects of free flow of capital, as now users can use their capital on multple blockchains. As of June 2022, the total value locked is still significant, as Figure 1 shows. With more than 40 bridging projects as of September 2021 [6], the trend is for projects to either mature by improving their security and usability or to disappear.

Although BI has the potential to enhance the current user experience in various dimensions, it does not come for free. To study its trade-offs, by formalizing the interactions between different systems (which we refer to as *domains*), we refer to the concepts of *cross-chain transaction* (*cctx*), *cross-chain logic* (or *cross-chain rules*), and *cross-chain model* (*ccmodel*). These concepts are important for reasoning about multi-chain applications: a *cctx* is a set of transactions abstracted into a logical unit of work [7], or a single atomic transaction [8]. A *ccmodel* is the set of rules that define conditions for *cctx*s plus a state (cross-chain state). The set of *cctx*s may follow a *ccmodel*, leading to *valid cross-chain state*, or not. If transactions do not follow the specified rules the *ccmodel* defines, the model is under-specified, or there is "suspicious" behavior (e.g., malicious, such as an attack, or non-malicious, such as a software bug). Effectively, a *ccmodel* allows one to have a baseline of expected behavior to compare ongoing *cctx*s with the baseline model. For instance, *ccmodel*s allow expressing complex cross-chain logic without having the protocol designer focus on timeouts, missing or corrupted information, and the technicalities of ad-hoc protocols. This allows the designer to focus instead on the business logic and its monitoring and achieve a separation of concerns.

The presented concepts have implications for understanding attacks on bridges. Some examples of recent mediatic attacks include the Wormhole bridge, where the attacker stole around $325M [9], [10], and the largest on-chain attack in the cryptocurrencies history, the Axie Infinity's Ronin Bridge [11], which caused around $625M in loss. In February 2022, the Wormhole bridge was attacked and resulted in $320M in damage [12]. In June 2022, the Harmony bridge was hacked, resulting in $100 million in losses [13]. Although the hackers were offered $1 million to return the funds to the community, it seems they have not complied [14]. In August 2022, the Nomad bridge collateral was stolen, resulting in the loss of $200M [15], despite the bridge being developed by an expert team and its being audited multiple times.

Looking at the facts, many of the largest decentralized finance hacks in blockchain history were performed in bridges [16], in a grand total of more than $1.5B in damages [17], [18]. The facts show that the community still does not know how to implement secure bridges, leading to systematic attacks on bridges, and damaging entire blockchain communities. The trend for attackers to exploit bridges will likely not disappear soon, as the more value bridges they hold, the more incentive criminals will have to attack those systems [19]. Capturing cross-chain logic for bridges would be useful to formalize the protocols (and help identify bugs and bottlenecks), monitor them, and act upon certain triggers. For instance, if an attack on a bridge is detected, a monitoring smart contract may pause the withdrawals, limiting the scope and impact of the attack. However, defining cross-chain logic is difficult because the base systems to be dealt with are heterogeneous and decentralized, and the systems built on top of them (e.g., decentralized applications) may have arbitrarily complex business logic and can be composed with multiple other systems (e.g., smart contracts). In a cross-chain setting, automating the discovery of *ccmodel*s and enabling its monitoring becomes very challenging, as there is a lack of tools to secure and monitor cross-chain applications. This is where our work fills the gap in current knowledge.

In this paper, we propose `Hephaestus`, a system that creates *ccmodel*s for fine-grain monitoring and auditing multiple blockchain use cases. Our system uses and extends a state-of-the-art BI solution, Hyperledger Cactus [20]. We build a *ccmodel* from *cctx*s formed by cross-chain events. Each model captures the rules that dictate which *cctx*s should be issued within a particular protocol. After that, such a model can be instantiated and capture a cross-chain state in real-time. The cross-chain state allows capturing relevant metrics of liveness (latency, throughput), safety (compliance with the transactional flow of the model), and others (cost, carbon footprint). This granularity allows us to answer several questions: *What is happening on the chains given a use case/protocol, at each moment?*, *Are there unexpected behaviors, i.e., deviations from the model?*, *What are the current bottlenecks of my cross-chain use case?*, and *Is there suspicious behavior concerning my use case?*. We pave the way for developers to anchor fail-switches to their use case based on some condition.

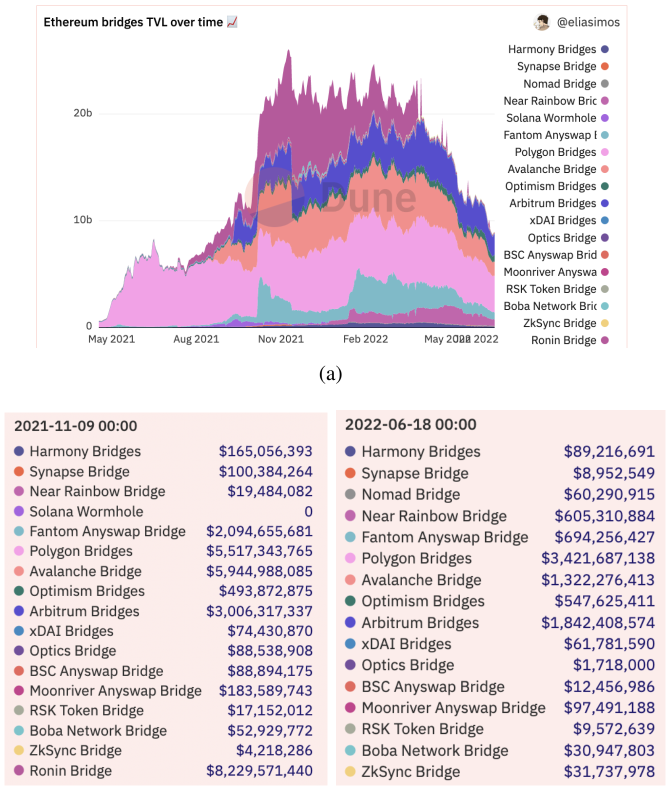
We validate these contributions by implementing a bridge system between Hyperledger Fabric and Hyperledger Besu and generating a *ccmodel* of its operations. As a technical contribution of independent interest, we have developed and improved various Hyperledger Cactus components over the last months, including the Hyperledger Fabric connector, the Hyperledger Besu connector, several test ledgers, the RabbitMQ test server, and several Python notebooks supporting our system. We empirically evaluate our system and its subcomponents according to a set of metrics on different scenarios and workloads.

II. Background

This section presents the background necessary to understand the paper, that is, processes, BI, and *cctx*s.

A. Process Mining Background and Applications

Understanding core concepts around processes is important to construct a system that can analyze *cctx*s and thus create *ccmodel*s. A process is a set of activities (or tasks) that aims to fulfill a certain goal [22]. For example, behind running a proof-of-stake blockchain, we have different processes a validator needs to run to achieve the end goal, the network's maintenance process, the consensus process, and so on.



(a)

2021-11-09 00:00	
● Harmony Bridges	$165,056,393
● Synapse Bridge	$100,384,264
● Near Rainbow Bridge	$19,484,082
● Solana Wormhole	0
● Fantom Anyswap Bridge	$2,094,655,681
● Polygon Bridges	$5,517,343,765
● Avalanche Bridge	$5,944,988,085
● Optimism Bridges	$493,872,875
● Arbitrum Bridges	$3,006,317,337
● xDAI Bridges	$74,430,870
● Optics Bridge	$88,538,908
● BSC Anyswap Bridge	$88,894,175
● Moonriver Anyswap Bridge	$183,589,743
● RSK Token Bridge	$17,152,012
● Boba Network Bridge	$52,929,772
● ZkSync Bridge	$4,218,286
● Ronin Bridge	$8,229,571,440

(b)

2022-06-18 00:00	
● Harmony Bridges	$89,216,691
● Synapse Bridge	$8,952,549
● Nomad Bridge	$60,290,915
● Near Rainbow Bridge	$605,310,884
● Fantom Anyswap Bridge	$694,256,427
● Polygon Bridges	$3,421,687,138
● Avalanche Bridge	$1,322,276,413
● Optimism Bridges	$547,625,411
● Arbitrum Bridges	$1,842,408,574
● xDAI Bridges	$61,781,590
● Optics Bridge	$1,718,000
● BSC Anyswap Bridge	$12,456,986
● Moonriver Anyswap Bridge	$97,491,188
● RSK Token Bridge	$9,572,639
● Boba Network Bridge	$30,947,803
● ZkSync Bridge	$31,737,978

(c)

Fig. 1. Total value locked, in dollars, for the main Ethereum bridges. Visualization created by Dune/eliasimos [21]. Figure 1a) shows the total value locked in dollars between May 2021 and June 2022. Figure 1b) shows the total value locked as of the 11th of November, 2022. Figure 1c) shows the total value locked on the 18th of June 2022.

The techniques for creating, analyzing, and optimizing processes are called process mining techniques [23]. Process mining has two sub-areas that help us in our endeavors: process discovery and process conformance. Process discovery aims to infer a process from an event log, this is, from a set of related entries, typically represented in a table. Entries in this table are *events*. An event is an occurrence targeting an activity and a point in time and is related to each other using a *case id*. Events point to an *activity* at a certain time, i.e., they have a *timestamp*. Activities are the operations that are executed within a process. Formally, an event e is a tuple $(act, caseId, timestamp, store)$, where act is the activity name, $caseId$ is the unique reference to the event, $timestamp$ refers to when the event was created, and a key-value store $store$. The key-value is in the form $\{(a_1, v_1), \ldots (a_n, v_m)\}$, where each a is an attribute of the event and v its value. The set of all events is $\mathcal{E}$.

The execution of a process produces what is called a *trace*, an ordered list of events with the same case id. Formally, a trace is a non-empty sequence $[e_1, \ldots e_n], \in [1..n], e_i \in \mathcal{E} \wedge \forall i, j[1..n], e_{i.caseID} = e_{j.caseID}$. An event log is a collection of traces referring to one or more cases. Discovering a process model can be done in various ways. For a detailed overview of how to generate process models, please refer to

[24]. Process conformance checks if the incoming transactions, including their ordering (or event entries), conform (are expected) to an existing model, helping evaluate a property called replay fitness. Conformance is part of process monitoring, helping identify errors or deviations from expected behavior. Processes have different representations. Graphical representations include BPMN diagrams [25], a notation useful for complex process semantics. In BPMN, events are denoted as circles, activities as rounded squares, and gateways as diamond squares.

B. Blockchain and Interoperability

A blockchain or ledger $\mathcal{L}$ supports two basic operations: reads and writes. Keys index information on blockchains, i.e., we look at blockchains as key-value stores. A read operation obtains the value for a certain key, i.e., $\texttt{read}_{\mathcal{L}}(key) = value$. A write operation on a key updates the value and returns 1 if successful, otherwise it returns 0: $\texttt{write}_{\mathcal{L}}(key, value) \rightarrow \{0, 1\}$. We call these writes local transactions. The history of each key's values is conserved by the blockchain data structure, which aggregates transactions (write requests) into cryptographically signed blocks. Reads are used to capture the part of the state relevant for interoperability processes.

This simple functionality allows to execute local transactions and read the global state. BI is the problem of coordinating *local reads* and *local writes* such that they satisfy some cross-chain logic. This is, reads from $\mathcal{L}_1$ can be composed with a write-on $\mathcal{L}_2$, realizing multiple use cases, such as data transfers, asset transfers, or asset exchanges [26]. Extensive work has been done in this area, including using two-phase commit to provide *cctx*s ACID [27] properties, where each local transaction executes successfully, or none at all [8]. We assume there is a cross-chain protocol deployed that orchestrates *cctx*s. A cross-chain is an abstraction rooted in a set of local transactions from different systems (e.g., enterprise legacy system, centralized databases, blockchains), respecting a set of rules. Further ahead in the paper, we explain formally what these concepts are. We will map the concept of a *cctx* as a set of events (which represent local transactions) that constitute a trace over a process model. A local transaction is a transaction native to a given technological environment, called *domain*. Examples of domains are blockchains such as Hyperledger Fabric network, the Ethereum main net, a Substrate-based parachain, Optimism, centralized databases, and distributed databases. Transactions trigger state changes, this is $s \xrightarrow{tx} s'$. Transactions have different life cycles, data formats, and properties as a function of their domain.

From the Blockchain View Integration Framework [2], a local transaction t is a tuple:

$$t \equiv (tid, t, target, payload, \sigma_{K_s^p}(tid, t, target, payload)) \tag{1}$$

where tid is the local transaction id, t is the timestamp, $target$ is the state key to which the transaction points, $payload$ is the payload (e.g., smart contract call) that will yield a state change (or a new value for a certain state key), and $\sigma_{K_s^p}(tid, t, target, payload)$ is a signature of the issuer of the transaction. We denote the execution of a local transaction t (in terms of state changes) as $s(t)$. Each state has a key, s_k and a value $s_{k,v}$. Thus, $s(t) = (s_k, s_{k,v})$ represents the state value of state s after the execution of transaction t (and thus, $t.target = s_k$).

III. Cross-chain Transactions

In this section, we define *cctx*s and their atomic units, the *cross-chain events* (*ccevent*s).

A *ccevent* extends a local transaction with metadata. We consider this metadata to be a set of non-native attributes (or parameters) $\{a_1, a_2, ..., a_n\}$ and their values $\{v_1, v_2, ..., v_n\}$. A *ccevent* e has native attributes (e.g., *tid*, *target*, and other elements from the transaction defined in the previous section), and non-native attributes, obtained via a function add, i.e., $e = \texttt{add}_{t^d}(a, data)$, where *add* is a function that adds data item $data$ to an attribute a of a local transaction t from ledger l. Each $data$ item is a non-native parameter (marked with ✗ in Table I). The native parameters can be obtained from the underlying domains or systems, i.e., they can be retrieved from the nodes supporting the blockchains without post-processing. Non-native parameters are externally obtained and are used to enrich local transactions. Native parameters may be used to calculate non-native parameters. For example, the carbon footprint depends on native parameters (e.g., on the native parameter cost (gas), in Ethereum).

Parameter	Type	Native
case ID	string	✗
receipt ID	string	✓
timestamp	Date	✓
blockchain ID	string	✗
invocation type	string	✓
method name	string	✓
parameters	string	✓
identity	string	✓
cost	number	✓
latency	number	✓
carbon footprint	number	✗

TABLE I
ccevent PARAMETERS, THEIR TYPE, AND NATURE (NATIVE ✓ OR NOT ✗).

A *cctx* *cctx* is a tuple $(\mathcal{E}, \mathcal{R})$ of n ordered events (by time) $\mathcal{E}$ from a subset of domains (e.g., ledgers) $\{d^1, ..., d^n\} \in \mathcal{D}$, i.e., $\mathcal{E} = \{e_1^{d^1 \in \mathcal{D}}, ..., e_n^{d^k \in \mathcal{D}}\}$, and $\mathcal{R}$ is a set of rules. We refer to cross-domain transactions as *cctx*s where the involved systems are exclusively distributed ledgers. Rules define conditions that must be verified to each event within a *cctx* to be executed - they depict the dependencies of each event on, for example, global time, local state, and third party domain state. Formally, a rule is a datalog rule [28], [29]. A datalog rule contains a head $R_{\mathcal{E}}$ and a body, and it is defined recursively. Given a set of predicates $\zeta = \{\zeta_1, \zeta_2, ..., \zeta_n\}$ over a set of events $\mathcal{E}$, we have that, for a certain time interval t_δ a rule is given in the form $R_{\mathcal{E},t_\delta} \longleftarrow \zeta(\mathcal{E})$ (we omit t_δ for simplicity of representation). The event set satisfying $R_\mathcal{E}$ are the intersection $\{\mathcal{E} | \zeta_1(\mathcal{E}) \wedge \zeta_2(\mathcal{E}) \wedge ... \wedge \zeta_n(\mathcal{E})\}$, this is, for an event set to satisfy a rule, it needs to satisfy all predicates. Each predicate ζ can define the conditions over transactions, i.e., temporal dependencies, the domain of a transaction, or a target function. For example, consider the following rule (predicate set):

$$\zeta(\mathcal{E}) = \begin{cases} \zeta_1(e) = e^x \prec e^y & \text{order dependency} \\ \zeta_2(e) = \forall e : e^x \vee e^y & \text{included domains} \\ \zeta_3(e) = \exists e : e.cost < z & \text{event attributes} \\ \zeta_4(e) = e_w.target = e_{w+v}.target & \text{event payload} \end{cases} \tag{2}$$

In this predicate set, ζ_1 defines any event happening in the domain d_x precedes ($\prec$) any event happening in the domain d_y. Predicate ζ_2 defines that events can only be part of domains x or y. Predicate ζ_3 states that there is at least one event in the event set, so its cost is less than z. Predicate ζ_4 states that the target of a transaction repeats every v transactions. Other predicates can be set for any of the attributes of a *ccevent*, in Table I. While we require each event to satisfy each sub-predicate of ζ, we can also set the validity of rules as the union $\{\mathcal{E} | \zeta_1(\mathcal{E}) \vee \zeta_2(\mathcal{E}) \vee ... \vee \zeta_n(\mathcal{E})\}$, or any other combination of predicates. We assume that there is an efficient way to transform a set of conjunction predicates into disjunctions or other formats. We define a function `verifySatisfability` that takes a predicate and an event, and outputs 1 if the event satisfy the given predicate and 0 otherwise, i.e., $\texttt{verifySatisfability}(e, \zeta) \longrightarrow \{0, 1\}$. We can then use this predicate for each event to assert a rule's validity.

To understand how this concept applies to practice, consider the following (simplified) rule that dictates the necessary events for a valid cross-chain asset transfer:

$$\zeta'(\mathcal{E}) = \begin{cases} \zeta'_1(e) = \underbrace{(\forall_e(e^x \vee e^y)}_{\text{events happen in x or y}} \wedge \underbrace{\forall_{e \in (x,y)}(e^x \prec e^y)}_{\text{events on x happen before y}} \\ \zeta'_2(e) = \underbrace{e^x.target = exists(a) \wedge e^x.target = lock(a)}_{\text{asset can only be locked if exists}} \\ \zeta'_3(e) = \underbrace{\zeta'_2}_{\zeta_2\text{' is satisfied}} \wedge \underbrace{e^y.target = mint(a)}_{\text{a mint can occur in domain y}} \end{cases} \tag{3}$$

Let us define rule ζ, the disjunction of the ζ' predicate set. The predicate set ζ' defines a set of conditions for a cross-chain asset transfer to be valid. First, as determined by ζ'_1, events in domain x must happen before events in the domain y. This paves the way for a lock on a source blockchain to be done before a mint on a target blockchain. Predicate ζ'_2 states that an asset from the source blockchain must exist before it is locked. Predicate ζ'_3 states that before an asset is minted on the target blockchain, predicate ζ'_2 must be satisfied. One could add more rules, such as the time for a mint transaction has to be done before block b, i.e., $e.target = mint \wedge e.timestamp \prec b$. We illustrate a cross-chain use case that allows asset transfers, in finer detail, in Section VII.

Cross-chain rules are enforced by constructs such as smart contracts. The combination of smart contracts in different domains realizes and then populates a cross-chain state. As each domain has its clock, time is not necessarily equivalent to other domains (e.g., time in blockchains, typically measured in block height, is different in Ethereum vs. Bitcoin). Thus, we need a global clock, from an external observer, in this case, the *ccmodel* generator, that could interpret time similarly. There is a clock that observes a series of blockchains and attributes a timestamp to each other. Each local time can then be mapped to a global time.

IV. Cross-Chain Models

In this section, we introduce the notion of *ccmodel*. A *ccmodel* $\mathcal{M}$ is a tuple $(\mathcal{R}, cctx, map, \mathcal{S},)$, where $\mathcal{R}$ is a set of cross-chain rules, *cctx* is a set of *cctx*s, *map* is a function mapping a *cctx* to a set of events e, i.e., $map(cctx) \longrightarrow e$, and $\mathcal{S}$ is the cross-chain state. The state allows representing on-chain behaviour from observed events (built from local transactions).

Each model is a container for related *cctx*s. Each *cctx* has a set of rules that allow the verification of its correct execution: one can take a *cctx* and its rules and verify if the local transactions follow each rule. We say a *cctx* follows a cross-chain rule if the evaluation of each predicate on the combination of events forming a *cctx* is 1, i.e., $\texttt{valid}(cctx, rule) : \forall e \in cctx : \texttt{verifySatisfability}(e, rule) = 1$. The cross-chain state is a key-value store that holds attributes relevant to the cross-chain use case, i.e., they are defined on a case-by-case basis. We call some attributes of the *ccmodel metrics*. Metrics are performance attributes of a set of *cctx*s and provide meta-information about a cross-chain use case.

A. Properties

A *ccmodel* should provide two main properties.

- Probabilistic-Completeness: the larger the event log (i.e., the number of observed events and consequently *cctx*s), the higher the model completeness probability.
- Replay fitness: given an observation of the real-world use case, the matching between the observations (in the form of events) and the *ccmodel* is higher than a threshold probability p.

Completeness is related to precision. A precise model avoids underfitting, a degree of measurement of how complete is the *ccmodel*. The replay fitness expresses the ability to explain on-chain behavior, i.e., how close the *ccmodel* is to reality. Other properties that are interesting in our context matter but will be explored in future work. Those generalizations measure if the model is too tied to specific execution instances of a cross-chain use case. Simplicity measures if the model is understandable by humans. Other aspects are omitted from the model generation, such as the task of minimizing noise, that is, minimizing behavior that is infrequent and does not represent the typical behavior of the process.

B. Metrics

Within $\mathcal{S}$, we define metrics, $M = M_1, M_2, ..., M_n$. These metrics indicate points of interest in a cross-chain use case. Metrics realize a meta state, where metrics about the formation and execution events that lead to that state are created.

M_1: Latency: We define latency as the time between a local transaction (via extended clients) and the creation of an *ccevent*. The total latency of a *cctx* ($\delta(cctx)$) is given by the latency of each event $\delta(e)$ from each local transaction, summed to the operational latency of the *ccmodel* generator ($\delta(op)$):

$$\delta(cctx) = \sum_{\substack{i=1,...,n\\ j=1,...k}} \delta(e_i^{d_j}) + \delta(op) \qquad \forall d\, e \in cctx \tag{4}$$

The operational latency is the time the model generator takes to retrieve and process the local transactions.

The latency of a *ccmodel* is the sum of the latency of each *cctx*:

$$\delta(\mathcal{M}) = \sum_{i=1,...,n} \delta(cctx_i) \tag{5}$$

M_2: Throughput: The throughput of a *cctx* is defined as $\frac{1}{\delta(cctx)}$, and it counts the number of sets of events processed per unit of time. Effectively, the latency for each event compresses the issuance and processing of each local transaction (which can take a long time depending on the blockchain), plus operational costs. The slowest finalization time δ_{fmax} can be a useful metric to complement throughput (and help identify bottlenecks in a *cctx*).

$$\delta_{fmax} = \max_{e_i \in \mathcal{E}, d \in \mathcal{D}} (\delta(e_i^d) \tag{6}$$

M_2: Carbon footprint

A carbon footprint of a *cctx* is the total amount of carbon dioxide associated with its preparation, execution, and commitment[1]. In this section, we make an educated guess for each system supported in our examples: a Hyperledger Fabric network (private blockchain) and a Hyperledger Besu blockchain (private blockchain).

Proof-of-work blockchains, such as the Bitcoin blockchain, are said to be using 130TWh (terawatts hour) of energy per year (comparable to a country like Ukraine or Argentina) [30], each transaction could account for 830kWh, or 3.6×10^{-1} tCO_2/kWh (metric tons of CO_2 per kilowatt-hour) [31], assuming all the nodes are being operated in the United States [32]. Translating kilowatts per hour to carbon emissions is difficult because the rate depends on the energy sources, the hardware the nodes run, and the machine (cloud vs. local). For Ethereum, the associated energy expenditure would be around 50kWh per transaction, an estimate of 2.165×10^{-2} metric tons CO_2/kWh. The energetic expenditure associated with all transactions in a year would be around 26TWh.

Tezos, a proof of stake blockchain, consumes 0.00006 TWh, or 0.6 KWh per year or 0.030 Wh per transaction [31]. However, other estimates put Tezos using around 41 Wh per transaction [**?**]. Each transaction could account for 1.3×10^{-7} metric tons CO_2/kWh or 1.8×10^{-4} metric tons CO_2/kWh, depending on the estimate.

To our knowledge, there are no studies on the energetic consumption of Fabric or Besus' private networks. Since their consensus is pluggable, using RAFT, or IBFT, respectively, we expect the energy expenditure to be in the same order of magnitude as Proof-of-stake systems or less. Therefore, we use the Tezos blockchain estimation, 1.8×10^{-4} tCO_2/kWh per transaction, knowing that this would be an upper limit. The total carbon footprint originated by a *cctx* in a system composed of private blockchains is then given by:

$$carbon = constant \times |cctx|(tCO_2/kWh) \tag{7}$$

where the constant 1.8×10^{-4}, and $|cctx|$ is the number of events in a *cctx*.

M_3: Cost and Revenue: Each local transaction might have a cost of transaction fees plus operation fees (in case a relayer or entity is transporting the local transaction payload across chains). Inspired by [33], we define the cost c of a *cctx cctx* events as the sum of variable costs (c_δ) plus operational costs (c_{op}):

$$c(cctx) = \sum_{\substack{i=1,...,n\\ j=1,...k}} cost_\delta(e_i^{d_j}) + c_{op} \qquad \forall d\, e \in cctx \tag{8}$$

The environment (e.g., via our system) typically gives information about these costs. The revenue is calculated in a way similar to the above formula. We can then calculate the profit of each *cctx* by subtracting the costs from the revenue. The concept of revenue can be modeled as positive utility and cost as negative utility, which sometimes maps better to real-world applications.

V. Hephaestus: A Cross-Chain Model Generator

`Hephaestus`[2] is a software system that generates *ccmodels* by mapping local transactions to *ccevents* (events) and then processing them, generating *cctxs*. Those *cctxs* assist generating a *ccmodel*, which holds metrics of interest. We assume the existence of several domains that can emit events (step ① from Figure 2). Furthermore, the cross-chain logic is realized by a cross-chain protocol enforced by smart contracts on both domains.

After defining our domain scope, a set of modified blockchain clients, called *connectors* issue transactions against target blockchains via an end-user or application (step ②). These blockchains emit events (or transaction receipts) that our connectors capture. `Hephaestus` then collects the local transactions (also called transaction receipts) in step ③

[1] A detailed and rigorous analysis of the carbon emission of different blockchain platforms is out of the scope of this work.

[2] Hephaestus is the greek god of metallurgy (that can connect different chains into a single useful artifact.)

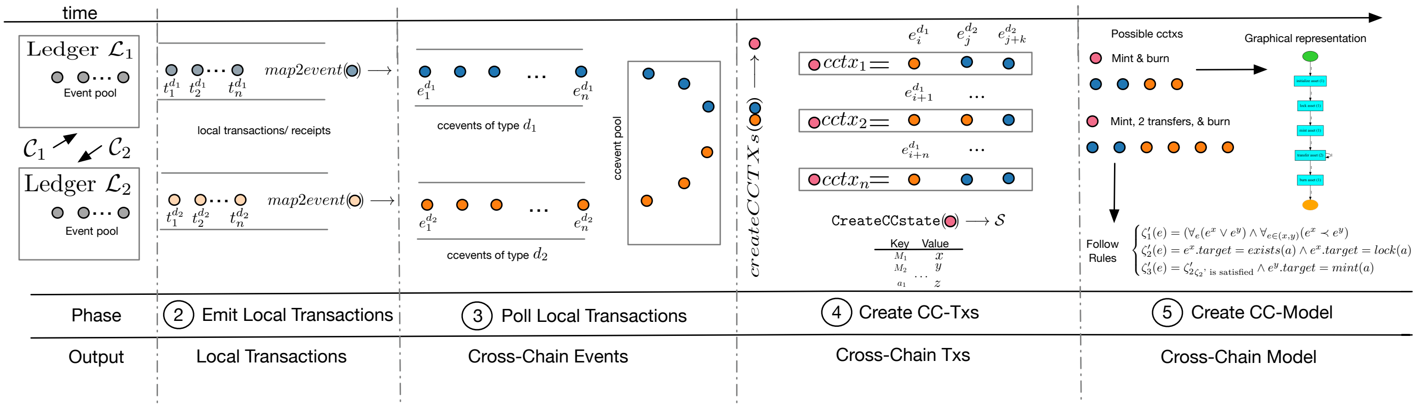

time
Ledger $\mathcal{L}_1$
Event pool
$\mathcal{C}_1$
$\mathcal{C}_2$
Ledger $\mathcal{L}_2$
Event pool
$t_1^{d_1}$ $t_2^{d_1}$ $t_n^{d_1}$
$map2event(●) \longrightarrow$
local transactions/ receipts
$t_1^{d_2}$ $t_2^{d_2}$ $t_n^{d_2}$
$map2event(●) \longrightarrow$
$e_1^{d_1}$ $e_n^{d_1}$
ccevents of type d_1
ccevent pool
$e_1^{d_2}$ $e_n^{d_2}$
ccevents of type d_2
$createCCTXs(●) \longrightarrow$
$e_i^{d_1}$ $e_j^{d_2}$ $e_{j+k}^{d_2}$
$cctx_1 =$
$e_{i+1}^{d_1}$
$cctx_2 =$
$e_{i+n}^{d_1}$
$cctx_n =$
CreateCCstate(●) $\longrightarrow \mathcal{S}$
Key Value
M_1 x
M_2 y
a_1 z
Possible cctxs
Mint & burn
Mint, 2 transfers, & burn
Graphical representation
initialize asset (1)
lock asset (1)
mint asset (1)
transfer asset (2)
burn asset (1)
Follow Rules
$\zeta_1'(e) = (\forall_e(e^x \vee e^y) \wedge \forall_{e \in (x,y)}(e^x \prec e^y)$
$\zeta_2'(e) = e^x.target = exists(a) \wedge e^x.target = lock(a)$
$\zeta_3'(e) = \zeta'_{2_{\zeta_2}\text{' is satisfied}} \wedge e^y.target = mint(a)$
Phase
② Emit Local Transactions
③ Poll Local Transactions
④ Create CC-Txs
⑤ Create CC-Model
Output
Local Transactions
Cross-Chain Events
Cross-Chain Txs
Cross-Chain Model

Fig. 2. Cross-chain model pipeline, spanning from phases 2 to 5.

and generates *ccevents* enriched with metadata. After that, it generates a set of *cctxs* (④) and finally it generates a *ccmodel* (⑤). The next section will illustrate these last three steps in finer detail. Business logic can be defined to facilitate the integration with legacy systems or to implement audit or monitoring functionality. Finally, an end-user should be able to query real-time metrics using a dedicated dashboard.

A. System Model

Cross-chain applications are structured as multi-step protocols, where there are different types of agents. Agents are users (e.g., end-users, relayers [2], or protocol administrators), and smart contracts. Users take turns doing off-chain processing and interacting with one or more domains (e.g., submit transactions against smart contracts). Agents are considered Byzantine, i.e., they can attempt to deviate from a protocol. Smart contracts enforce cross-chain logic. Specifically, smart contracts running on different blockchains are trusted replicas in the state-machine replication literature (similarly, each domain is considered trusted, even if centralized).

This model protects agents who honestly follow the protocol from those who do not. This assumption implies that if a domain cannot be trusted, then safety on cross-chain rule execution cannot be guaranteed. Domains can only learn the state of other domains and their changes using an agent. Of course, each domain must decide whether an agent is telling the truth. This can be achieved with a cross-chain protocol, where trust assumptions vary substantially [1]. Our model assumes a cross-chain protocol that can provide safety and liveness. Safety states that the execution of cross-chain rules results in a consistent cross-chain state for a certain definition of a cross-chain state. In a cross-chain asset transfer, safety would mean that there is no double spend, this is, it is not possible to mint an asset on a target blockchain without first locking it on the source blockchain; and similarly, it is not possible to unlock such asset on the source blockchain without first burning it on the target blockchain. Liveness ensures that all cross-chain rules from a protocol are evaluated (executed). In the case of an asset transfer protocol, liveness means that "conforming parties' assets cannot be locked up forever".

`Hephaestus` observes the interactions between users and smart contracts. We assume a partial synchrony model, i.e., there is some finite unknown upper bound δ on the creation of events within domains. The bound δ is not known in advance and can be chosen by the adversaries (in each blockchain). We consider the delay to be $\max(\delta_{d1}, \delta_{d2}, \ldots, \delta_{dn})$. `Hephaestus` runs a global clock, i.e., it can measure time against different domains, despite their clocks being different. `Hephaestus` provides *accountability*, meaning It is possible to verify a set of cross-chain rules has been followed or not.

B. Cross-Chain Model Generation

In this section, we explain how to generate a model, focusing on phases ② to ⑤. Our pipeline is divided into phases: ② *Emit Local Transactions*, ③ *Poll Local Transactions*, ④ *Create CC-TXs*, and ⑤ *Create CC-Model*. Figure 2 represents the four phases, which we will describe in detail next.

In phase ②, we start by listening to local transactions in our domain set $\mathcal{D}$. Each domain in our system has an accessible event pool, from which we can fetch the events used to build the model. Without loss of generality, and to simplify our reasoning, we look for local transactions in the domains ledger $\mathcal{L}_1$ (●), and ledger $\mathcal{L}_2$ (●). The considered transactions are created and submitted by our connectors, $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Transactions have a case id, meaning local transactions without this special identifier are not considered. Our clients capture the relevant transactions and send them to the *ccmodel* generator, starting the next phase. In phase ③, the raw transaction receipts enter a processor module that translates local transactions into a *ccevent* according to a function $map2event(tx_{\mathcal{L}}) \longrightarrow ccevent$.

The output of phase ③ are *ccevents* coming from $\mathcal{L}_1$, ● $event_1$, and from ledger $\mathcal{L}_2$, ● $event_2$ which we aggregate onto a *ccevent* log, with events coming from different ledgers. Therefore, events implement a standardized data model. After constructing a set of events, we proceed to phase ④. In this phase, we receive an event log and output the cross-chain state and a set of *cctx*s (via a *createCCTX*s function). We create cross-chain transactions by aggregating *ccevent*s by its case id and calculating relevant metrics (please refer to Section VI). We build the cross-chain state according to Algorithm 1.

Algorithm 1: `CreateCCState`. Creation of a cross-chain state from a set of *ccevent*s

```
   Input: Set of events $\mathcal{E}$
   Input: State builder algorithm $\mathcal{A}$
   Input: Cross-chain rules $\mathcal{R}$
   Output: Cross-chain state $\mathcal{S}$
1  $\mathcal{S} \leftarrow \emptyset$
2  foreach $e \in \mathcal{E}$ do
3      if $\texttt{verifySatisfability}(e, \mathcal{R})$ then
4          continue
5      end
6      else
7          ...
8          rules are not satisfied, emit alert and abort abort
9      end
10     if $\nexists \mathcal{S}[e.caseID]$ then
11         each cross-chain state key is indexed by case ID
12         $\mathcal{S}[e.caseID].caseID$ = e.caseID use raw events
             data to setup cross-chain state
13         $\mathcal{S}[caseId].id$ = randomNumber()
14         smart contract function or target
           $\mathcal{S}[caseId].targetFunction$ = e.target
15         $\mathcal{S}[caseId].transactionListRefs$ = e.receiptID
16         $\mathcal{S}[caseId].lastUpdated$ = e.timestamp
17         $\mathcal{S}[caseId].latency$ = e.latency
18         $\mathcal{S}[caseId].cost$ = e.cost
19         $\mathcal{S}[caseId].revenue$ = e.revenue
20         $\mathcal{S}[caseId].callers$ = e.identity
21         $\mathcal{S}[caseId].carbonFootprint$ = e.carbonFootprint
22         ...
23     end
24     else
25         ...
26         if state key already exists, update its values (i.e.,
           calculate average latency, cost)
27     end
28 end
29 $\mathcal{S} = \texttt{processArbitraryState}_{\mathcal{A}}(\mathcal{S})$
30 return $\mathcal{S}$
```

Our algorithm receives the processed *ccevent*s and a processing algorithm $\mathcal{A}$. Each event follows the data model described in Section III. The processing algorithm is responsible for creating the part of the cross-chain state that is composesed of the metrics gathered from running several local transactions (lines 11-21, not meant to be exhaustive). If there are multiple *ccevent*s with the same case ID, it must be a *cctx* composed of multiple local transactions. The algorithm $\mathcal{A}$ adds the semantics of a multi-chain use case. We exemplify $\mathcal{A}$ by setting it to be a `verifyLock` function. This function would verify if an asset was locked on a source blockchain so it can be minted on a target blockchain, implementing the cross-chain state (basically one bit for each tracked asset, allowing to verify if it is locked on a source chain). Note that this function is illustrative and does not reflect a complex lock-unlock mechanism. For instance, the algorithm should check if an unlock with a newer timestamp happened regarding a locked asset. This function is encoded in Algorithm 2. Having a cross-chain state, we initiate the *ccmodel* generation phase ⑤. In this phase, we generate a *ccmodel* using an algorithm $\mathcal{A}$ and the *ccevent* log. Note that it is possible to additionally use information from the *cctx* log, such as the different metrics, to generate the model. Several graphical representations are possible, such as a process tree or a BPMN model (shown later in this paper).

Algorithm 2: Verification of a lock transaction referring to asset a

```
Input: Cross-chain state $\mathcal{S}$
Input: Case id $id$ referring to the lock
Output: Updated cross-chain state $\mathcal{S}'$
$\mathcal{S}' \leftarrow \emptyset$
foreach $s \in \mathcal{S}$ do
    if $s.caseID = id$ then
        if
          $e.target == \texttt{verifyLock} \wedge e.parameters == a$
          then
            s.lockedAssets[a] = 1
        end
    end
end
return $\mathcal{S}'$
```

C. Identifying non-conformance

In this section, we explain how we detect non-conformance behavior. Non-conformance behavior can be one of three: outliers, malicious intent (bug exploitation/attack), or non-modeled behavior. The baseline for detecting non-conformance is a *ccmodel*, which corresponds to a specification of expected behavior. We define a set of traces belonging to the set of all possible traces $\{t_1, ..., t_n\} \in \mathcal{T}$ as a current execution of a cross-chain use case. For each trace being executed, we consider a set of steps $s_1, ..., s_n$. We then take the sequence of steps and perform alignment. Alignment-based replay aims to find one of the best alignment between the trace and the model. Each alignment creates a set of pairs (trace, transition) such that for each pair, one of the following can occur: 1) *SYNC MOVE* 🟢 - both the trace and the model advance in the same way during the replay, meaning we have a match, 2) *MOVE ON LOG* 🔴 - the trace that is not mimicked in the model. This means there is a deviation between our specification and the observed behavior.

The idea is now to retrieve incoming *ccevent*s and build a trace. If a trace is *SYNC MOVE* 🟢, then it is common behaviour (expected). Otherwise, it is non-modeled behavior, *MOVE ON LOG* 🔴, which should be investigated. This behavior can result from under-modeling or malicious behavior (for example, an attack). In any case, the end-user may inspect the event leading to the trace and understand which parameter has caused such behavior. Malicious behavior can come in different forms. The most common are smart contract vulnerabilities holding the business logic that realizes the use case. Many more attack vectors exist, such as smart contract framework vulnerability, dependency vulnerability, cryptographic vulnerability, network attacks such as denial of service or network partitioning, consensus manipulation, and others [34]. In this paper, we focus on attacks on smart contract exploitation (in the form of manipulating a defined transaction flow). However, we could generalize our scheme to cover different attack vectors.

VI. Implementation

In this section, we present the implementation. The code is available on Github[3]. We developed our work as a Hyperledger Cactus (Cactus) [20] plugin. Cactus is a blockchain integration project supported by enterprises such as Blockdaemon, Accenture, IBM, Fujitsu, with more than 230 stars and 65 contributors. Next, we detail this paper's relevant technical contributions and the implementation of `Hephaestus`.

A. Connectors

We implemented two blockchain connectors to connect to multiple blockchains and retrieve transactions. Connectors are self-contained application programming interfaces that constitute the basis for interoperability functionality. The first connector binds our software to Hyperledger Fabric 2.2 — a permissioned blockchain system. Fabric is designed for enterprise-grade applications that benefit from decentralization. It supports smart contracts (called chaincode), that can be written in several general-purpose programming languages. The nodes execute proposals for transactions signed and sent to an orderer node. Orderer nodes reach consensus on the order of transactions, batch them into blocks, and link them, creating the blockchain. Then, new blocks are sent to the nodes on the network. Fabric has a key-value store that holds the most up-to-date values from the blockchain - for performance reasons; it allows chaincodes to retrieve state without reconstructing the blockchain. We implemented this connector, package name *cactus-plugin-ledger-connector-fabric* in Typescript, counting 4958 lines of code. We wrote 16 integration tests, accounting for 4450 lines of code. The connector supports functionality to issue transactions, deploy smart contracts, send transaction receipts to `Hephaestus`, and several administrative tasks (such as registering a new user).

The second connector connects to a Hyperledger Besu (Besu) 1.5.1 network. Besu is an open-source Ethereum client, that also has capabilities to span private networks. It allows for interacting with Ethereum networks, including participating in the consensus process, and to develop and deploying smart contracts and decentralized applications. Besu implements proof of authority algorithms such as IBFT (more suitable for private networks) and proof of work (Ethash). We implemented this connector, package name *cactus-plugin-ledger-connector-besu* in Typescript, counting 4629 lines of code. We wrote 14 integration tests, accounting for 3426 lines of code. The connector supports functionality to issue transactions (signed and unsigned), deploy smart contracts, send transaction receipts to `Hephaestus`, and several administrative tasks (such as obtaining a raw block from the network).

Concept	Implementation	Lines of code
Domain	Fabric ($\mathcal{L}_1$), Besu ($\mathcal{L}_2$)	–
Domain logic	Bridge smart contracts	819
Ledger client	Fabric ($\mathcal{C}_{\mathcal{L}1}$), Besu ($\mathcal{C}_{\mathcal{L}2}$)	17,463
Test ledger	Fabric and Besu test ledgers	1,873
Model Generator	`Hephaestus`	5,921
Process Discovery	pm4py	–
Process Conformance	pm4py	–

TABLE II
Implementation effort as the number of lines of code created, for each presented component

B. Test ledgers

We implemented tools to programmatically create test networks for Fabric and Besu, allowing for reproducible tests and debugging of our application, namely the tools *besu-all-in-one* and *fabric-all-in-one*. These tools not only allow the reproducibility of our work but also ease the developers to create new applications and build on top of `Hephaestus`. The all-in-one test ledgers are divided into two parts: 1) a test ledger manager, a program that launches, administrates, stops, and destroys test ledgers, and 2) Dockerfiles defining the networks. The Fabric test ledger manager has 1445 lines of code and, since deployed, had more than 80 thousand pulls from Dockerhub[4]. The Besu test ledger manager has 428 lines of code and, since deployed, had more than 390 thousand pulls from Dockerhub. Other test ledgers such as *corda-all-in-one* and *substrate-all-in-one* are available for the research community.

C. Smart Contracts

We implement a use case composed of an asset transfer across a Hyperledger Besu network and a Hyperledger Fabric network as a foundation for testing `Hephaestus` capabilities. A cross-chain asset transfer generates a set of events representing an asset lock on a source blockchain (Besu), and an asset unlock on a target blockchain (Fabric). This lock-unlock mechanism assures that the representation of a minted asset is pegged to a locked asset. In asset transfers, there is typically a third party actor called a relayer, which carries the proof of a lock to the target blockchain, so the mint can occur[5].

The use case is implemented as follows: on the Besu network, we have a Solidity smart contract with two methods: *create asset*, and *lock asset*. First, a user must create an asset and then lock it. On the Fabric network, the Typescript smart

[3] https://rafaelapb.page.link/code

[4] also accounts for pulls generated by the continuous integration pipeline, such as in automated tests

[5] We could model the relayers behavior in our use case, by adding two activities: *submit proof*, which contains a payload that certifies that an asset was locked from the source chain, and *proof submission*, a payload consisting of a proof that validates the minting of an asset. These events would be added by a system other than the mock blockchain (e.g., relayer).

contract allows a user to call *mint asset*, creating the representation of the locked asset. Using a cross-chain protocol, we assume that a relayer submits the necessary proofs on the Fabric blockchain for the mint to happen. After that, a user can freely transfer that token to other users in exchange for other tokens, using *transfer asset* (optional). Finally, if users want to recover the original tokens, they run *burn asset representation*. We assume that a relayer carries the necessary proofs of the burn to the source blockchain. This procedure would unlock the assets on the source chain.

D. Hephaestus

We implemented `Hephaestus` as a business logic plugin for Hyperledger Cactus, written in Typescript. Its latest version is version commit *8d8567e* (*stable branch)*, package name *cactus-plugin-cc-tx-visualization*. `Hephaestus` is initialized with a set of connectors used to capture local transactions. Then transactions are mapped to *ccevent*s. The *ccevent*s are used to build a cross-chain state, and then we give the *ccevent*s to a Python script (model generator) that, on its end, generates the *ccmodel*. The model generator used the open-source library pm4py version 2.2.20 [35]. We generate our model using the Inductive Miner algorithm [36]. Our plugin counts a total of 1492 lines of code. We wrote 18 integration tests, accounting for 4429 lines of code. To identify misconformance, we used an alignment technique, available in the *conformance_diagnostics_alignments* function from the *pm4py* library, namely the Scipy linear solver tool.

VII. Evaluation

Goals: The goals of the experiments are as follows. 1) evaluate `Hephaestus` performance in terms of transaction throughput, latency, and storage required. We also evaluate the scaling capabilities concerning the number of local transactions, activities, and domains. Goal 2) is to evaluate the system's capability to identify misconformance, given a baseline *ccmodel*

Experimental Setup: We deployed an instance of `Hephaestus` on Google Cloud (CPU with eight cores, 32Gb of RAM, SSD). The different event providers are our Hyperledger Fabric connector, and the Hyperledger Besu connector (version 1.0.0). We initialize a RabbitMQ server serving our event collector *rabbitmq-test-server*. Event emitters on the connectors are implemented as RabbitMQ clients. Every experiment was run 50 times (where we removed the first and last 10 runs, considering a total of 30 runs), and we report the average result, along with the standard deviation, for each run. We share the scripts to generate the plots and *ccmodel*s, making the evaluation process reproducible. Furthermore, we save the output of each evaluation scenario [6] and the generated cross-chain logs [7] and share it with the reader.

Metrics and Workloads: For each run, we capture the following metrics: throughput (*cctx*s) and their latency, storage cost, i.e., performance metrics. We test two scenarios under variable workloads, which we present later in this section. We characterize each scenario as a tuple *(interoperation mode, number of blockchains, event type, and workload)*. The interoperation mode states what cross-chain feature we are testing, asset transfers, asset exchanges, or data transfers. While intuitive, for space limitations, we refer to [26] for a detailed explanation. The number of domains reflects the number of ledgers or other systems emitting events in the scenario, namely Hyperledger Fabric, Hyperledger Besu, or a mock blockchain (essentially, we only model the message transmission). Finally, each workload contains details on the number of events, activities, and domains in that scenario. We implemented a workload generator that produces events across different blockchains. Events are then captured by `Hephaestus`.

A. Baseline: Dummy Use Case with Test Receipts

In this section, we depict the evaluation of our system using a mock blockchain.

Interoperation Mode: Asset Transfer

Number of domains: 1

Event Type: Test (Mock Blockchain)

Workload: 6 events, 6 activities, *ccmodel* generation algorithm $\mathcal{A} =$ `inductive miner`

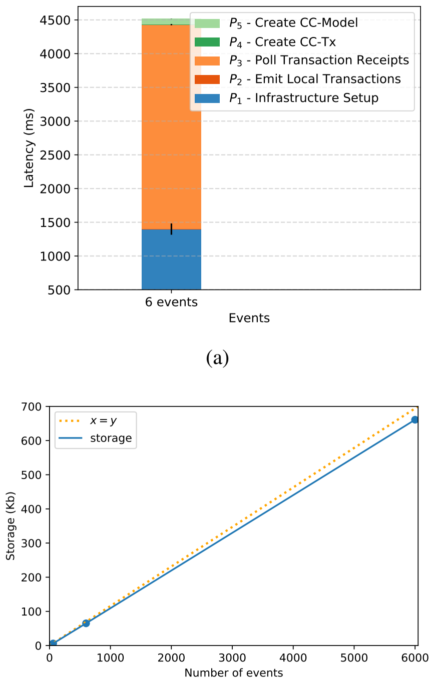
The dummy use case represents a *cctx* composed of 6 *ccevent*s. This transaction locks an asset from a source blockchain and unlocks a representation of the same asset on a target blockchain (typically using parties called relayers[8]). Instead of using blockchains to collect receipts, receipts are emitted by a single mock blockchain, which we call the *test blockchain*. The mock blockchain processes transactions as detailed in Section VI-C, namely *create asset*, *lock asset*, *mint asset*, *transfer asset* (optional) and *burn asset representation*.

We measure the performance of the following phases (see Figure 3a): the *Infrastructure Setup* (phase 1), the emission and polling of local transactions, as events *Emit Local Transactions* (phase 2.1), and *Poll Local Transactions* (phase 2.2), the creation of *cctx*s, *Create CC-Tx* (phase 3.1) and the creation of the *ccmodel*, *Create CC Model* (phase 3.2). The *infrastructure setup* includes setting the event emitters (connectors, including creating blockchain networks and initializing the connectors), setting the event collector (RabbitMQ server), and setting up `Hephaestus`. The *Emit Local Transactions* phase emits test events or issues transactions against the deployed ledgers. The *Poll Local Transactions* waits for the events and sends them to `Hephaestus` for processing. The *Create CC-Tx* generation includes mapping the local transactions to *cctx*s,

[6] Online: https://rafaelapb.page.link/cctx-viz-output

[7] Online: https://rafaelapb.page.link/cctx-viz-csv

[8] We could model a third-party responsible for carrying proofs of on-chain execution, the relayer [1] - yielding two more events, in addition to the modelled six. Those two extra events are modeled as activities: *submit proof*, which contains a payload that certifies an asset was locked from the source chain, and *proof submission*, a payload consisting in a proof that validates the lock of an asset and thus allows its mining.



(a)

(b)

Fig. 3. Figure a) shows the latency, in milliseconds, for each phase of the baseline test scenario. Figure b) storage requirements, in kilobytes, for a variable number of events.

and calculating *cctx* metrics. Figure 3a) shows the latency phase breakdown for the emission of six events. We can observe that the setup phase takes around 1.5 seconds, and the most time-consuming phase takes approximately 3 seconds, despite being mock transactions. Figure 4 shows the same breakdown for a variable number of events. Table III supports this figure by reporting the mean end-to-end latency for each phase along with the standard deviation. Phases 1 and 2.2 remain practically constant. Phase 2.1 is sublinear. Phase 3.2's performance indicates that after a certain threshold (between 600 and 6000 transactions), the system starts slowing down its performance.

We measure the storage required for generating, storing, and processing events into a *ccmodel*. Figure 3b) shows the required storage as a function of the number of events created. The RabbitMQ container and respective runtime data occupy 257Mb and 72.4kB, respectively. The storage requirements appear to be sublinear to the number of events - six events (one *cctx*) occupy 789 bytes, while six thousand events occupy around 6.6Mb. For a *cctx*, means each *cctx* occupies around 789 bytes + derived data (metrics, a few bytes). Since the metrics are five floats, a date, a string with 128 chars, and a list of events, each *cctx* occupies at least 937 bytes. Finally, the *cross-chain model generation phase* includes parsing the created *cctxs* and generating the *ccmodel*. The generated BPMN model for the dummy use case scenario is represented in Figure 5. Each *cctx* takes 985 milliseconds to build and has a carbon footprint of 0.

	Phase 1 🟦		Phase 2 🟧				Phase 3 🟩			
	——		Phase 2.1		Phase 2.2		Phase 3.1		Phase 3.2	
$events$	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
6	1397.53	83.06	0.47	0.51	3030.27	8.94	0.67	0.61	83.51	1.95
60	1387.93	20.09	2.20	0.66	3068.97	19.07	1.27	0.45	87.2	0.93
600	1388.33	22.26	13.03	3.02	3163.47	21.89	7.53	1.14	98.21	1.51
6000	1392.20	18.05	116.97	20.67	3459.37	18.36	26.5	3.42	265.61	4.18

TABLE III
END-TO-END PROCESS LATENCY MEAN (μ) AND STANDARD DEVIATION (σ), IN MILLISECONDS, AS A FUNCTION OF THE NUMBER OF EVENTS.

B. Use Case: asset transfer across heterogeneous networks

In this section, we depict the evaluation of our system using two blockchains.

Interoperation Mode: Asset Transfer

Number of domains: 2

Event Type: Hyperledger Fabric, Hyperledger Besu

Workload: 6 events (2 Besu plus 4 Fabric), 6 activities, *ccmodel* generation algorithm $\mathcal{A} = \texttt{inductive miner}$

Next, we illustrate an asset transfer between a private network running Hyperledger Besu and a private network running Hyperledger Fabric. The asset transfer process is the same as in the baseline scenario, i.e., a *cctx* is composed of 6 events. An `Hephaestus` instance is connected to a Fabric connector and a Besu connector. Each connector is connected to a Fabric network version 2.2 and Besu network version 21.1, respectively. The Fabric network consists of 2 peers and 1 orderer, using Raft as the consensus protocol of orderers and LevelDB to maintain the local storage in each node. The Besu network consists of a solo node network. The use case explored in this section follows the same transaction flow as the baseline, i.e., transactions *create asset*, *lock asset*, *mint asset*, *transfer asset* (two of them) and *burn asset representation* are issued in this order. This flow implements a simplified version of a cross-chain promissory note transfer [8] between Hyperledger Besu and Hyperledger Fabric. We used the smart contracts described in Section VI.

When testing the emission of 60 and 600 transactions, as expected, the Infrastructure Setup and Emit Local transaction phases take the most time. The median latency required for these phases is 151, 237, and 1115 seconds, respectively, for 6, 60, and 600 events. The infrastructure setup phase takes 90%, 56%, 12% of the overall execution latency, while the transaction emission takes 7%, 42%, 88%, for 6, 60, and 600 transactions, respectively. Since these phases take the most of the execution time, we illustrate the breakdown of the remaining phases, "Poll Local Transactions", "Create CC-Tx", and "Create CC Model", in Figure 4. The storage requirements are similar to the baseline use case. For a 60 event execution, we obtain that each *cctx* (6 events) takes 2,04 seconds to construct and has a carbon footprint of 3.85×10^{-5}.

C. Baseline Vs. Use Case

The bottleneck for both scenarios is the infrastructure setup (phase 1) and transaction emission phase (2.1) or polling

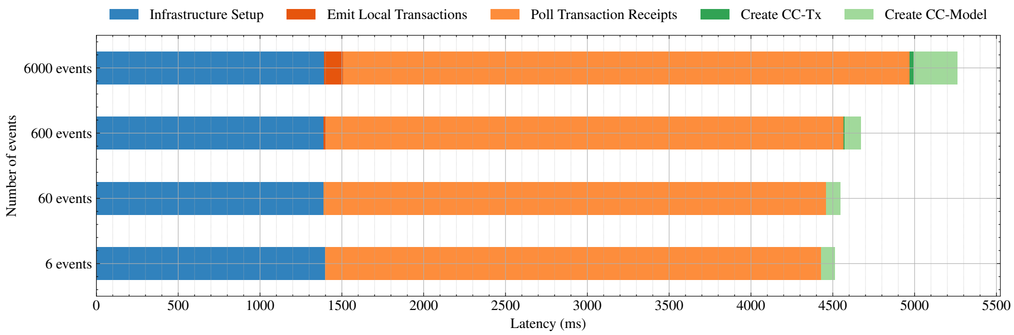


Fig. 4. Latency for each phase of the baseline test scenario, for a variable number of events.

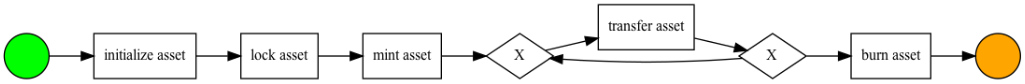


Fig. 5. Generated business process modelling notation model for the events generated in the baseline phase.

transactions (phase 2.2). For the use case, the bottlenecks are the infrastructure setup (phase 1) and receipt emission phases (phase 2.1). We observe that the infrastructure setup and transaction emission phases occupy around 97%, 98%, and practically 100% of the execution time, depending if we are emitting 6,60, or 600 events, respectively. This is expected, and we conclude the bottleneck is the transaction execution and commitment. In the dummy use case creating receipts is a cheaper task than retrieving them; in this use case, the inverse happens because executing transactions on blockchains is generally an expensive task. In a production environment, the cross-chain throughput is limited by the finality speed of the underlying systems: the infrastructural part of `Hephaestus` is efficient in issuing the transactions and retrieving the respective receipts. Varying the number of domains/blockchains should not affect the creation of *cctx* as all transaction receipts are interpreted as *ccevent*s. The complexity of transforming the receipts into events may vary significantly, but our experiments show a very low overhead. Furthermore, the setup phase only needs to be performed once. We conclude that our system is scalable in terms of latency and extensibility.

D. Identifying Misconformance

In this section, we run experiments that allow us to evaluate if `Hephaestus` can detect deviations from expected behavior. Expected behavior, or the specification, is given by the events we emit. After generating the *ccmodel*, we generate another set of events, this time in the following order: create asset, mint asset, transfer asset, burn asset. Note that the lock asset phase is not present - this emulates a user attempting to mint an asset without an appropriate lock. We obtain detailed alignment information about transitions that did not execute correctly, namely the mint before the lock. While it is possible to obtain a set of detailed metrics such as throughput and cost analysis of real-time flows, we stick to the conformance of process (the fitness) for the sake of space. Figure 6 shows the expected process for this use case. We obtain that a mint should occur after it has occurred (`'MintAsset', '>>'`), and fitness 82%. The trace generated by our implementation and the expected traces differ, originating a *MOVE ON LOG* 🔴.

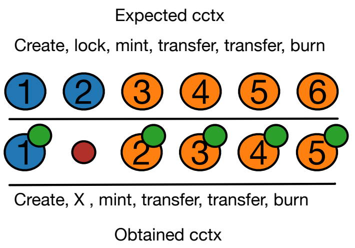

Expected cctx
Create, lock, mint, transfer, transfer, burn
1
2
3
4
5
6
1
2
3
4
5
Create, X , mint, transfer, transfer, burn
Obtained cctx

Fig. 6. Misconformance detection process. Domain 1 events 🔵 are happening on the source chain; Domain 2 events 🟠 happen in the target blockchain. Events can lead to a *SYNC ON MOVE* 🟢 or *MOVE ON LOG* 🔴.

E. Qualitative Analysis

`Hephaestus` contributes to mitigating bridge hacks by 1) generating a *ccmodel* of the bridge protocols, allowing reasoning about the protocol flow, bottlenecks, and possible threats and vulnerabilities, and 2) minimizing the attack consequences by finding active monitoring and detecting suspicious behavior in real-time. From Section VII-D, we conclude that the replay fitness of our model is 82%, although it is probably not complete due to the low number of traces captured. However, our tool can be deployed and set up to capture real use case events from deployed smart contracts in production. Our tool can be extended to act upon suspicious activity (e.g., freeze certain types of transactions) by implementing the following procedure: first, `Hephaestus` generates a *ccmodel*. Next, it listens for transactions on the interest domains, and a trace is built. Every step/transaction is analysed and classified as 🟢 or 🔴. An event is emitted if the transaction creates a discrepancy from the original model. This event triggers business logic that deals with the occurrence (e.g., requesting human confirmation; freezing a smart bridge contract). This sort of mechanism would probably help mitigate bridge hacks, and it is being actively explored by the industry [37]. For this, good *ccmodel* representations are needed. We generate BPMN models, that are good for expressing the semantics of a cross-chain use case graphically, but research is still lacking. An important assumption is that the model is complete, i.e., models all the desired behavior. However, this is not always the case, and some ● events can be false negatives. Creating robust models that tolerate noise and evaluating those models is an evolving, core challenge in the process mining area that would have repercussions in generating and maintaining *ccmodel*s [23].

In the baseline, we assume the *revenue*, *cost*, and *carbon footprint* to be zero. Note that these parameters can be adjusted according to the use case, allowing semantically enriching each transaction. Associating cost and revenue values to transaction receipts would help calculate capital profit taxes for a certain jurisdiction. In the use case scenario, we populate the values for carbon footprints of the test ledgers accordingly to Section IV. Although these numbers are only references, it provides a step toward better energy measurements and ecological responsibility. Finally, our system is designed modularly such that new blockchains can easily be supported. It has the potential to be integrated into cross-chain APIs for such purposes. The possibility of retrieving the cumulative metrics for all *cctx*s processed in the *ccmodel* allows enhanced and fine grain monitoring of cross-chain logic. We would like to emphasize that Hephaestus could be used for use cases other than bridges - we will address this topic as future work.

VIII. Related Work

Hephaestus is the result of an inter-disciplinary work that combines the fields of blockchain interoperability, on-chain analytics, and process mining applied to the blockchain.

Monitoring Blockchain Interoperability

The work by Zhang et al. [38] seems to be the most similar to ours. The authors create a tool to identify miss-conformance on the lock-unlock bridge mechanism. However, this work is directed specifically from bridges and not arbitrary cross-chain use cases. On the other hand, BUNGEE is a general-agnostic framework that inspires this work. In this paper, a tool that produces consolidated views over user activity on different blockchains [2] is proposed. Hephaestus can complement BUNGEE to generate metrics, protocol behavior patterns, and individual user activity. Hephaestus can be deployed over interoperability protocols such as ODAP/SAT [8], [39], [40], XCLAIM [41], and many others [1], to provide a monitoring layer.

On-chain Analytics

In the field of on-chain analytics, some industry solutions exist and are well-adopted: the Dune tool allows to Explore, create and share crypto analytics, including key metrics for DeFi, NFTs, and more, expressive queries, and the visualization of information in dashboards. Hephaestus would allow for the creation of a cross-chain Dune tool by cross-referencing transactions in multiple chains [42]. Chainanalysis provides a dashboard for investigation, compliance, and risk management tools to assert compliance with jurisdictions and fight fraud and illicit activities [43]. For example, it allows to visualize the flow of funds and track movements across currencies. Our tool would provide possibilities to port this monitoring for the cross-chain scenario. Metla finance [44], Morali [45], and Rokti [46] allows a unified view of user assets over different blockchains. Hephaestus would allow extending the views to support arbitrary states across blockchains. Certik provides a monitoring layer for analyzing and monitoring blockchain protocols and DeFi projects, but only from a security perspective [47]. Token Flow is the closest work to ours, an analytics tool to track cross-chain asset transfers [48]. However, Token Flow does not support arbitrary cross-chain use cases.

On the academic side, we have several tools that allow on-chain analysis of smart contracts for security purposes [34], [49], [50], performance [51], [51], [52], compliance and anti-fraud [53], and others [54], [55] . However, such projects provide a sort of meta-view over user activity, do not provide specific information about interaction with protocols, and are not generalizable.

Process mining on blockchain

In the process mining area, some work has been done to apply it to the blockchain. In [56], the authors used process mining techniques to specify the behavior of the Augur protocol, discovering bottlenecks and proposing improvements. Some tools to automatize the creation of process models from blockchain protocols to facilitate multiple goals have been proposed [57]–[60], but none for the cross-chain scenario.

IX. Concluding Remarks

The need for multi-chain applications introduces additional challenges to end-users and developers, including usability friction, lack of control over the cross-chain state, and security issues, in the form of a more extensive attack vector. We are witnessing the exploitation of a crescent attack vector on decentralized protocols. To address this problem, we propose Hephaestus.

Hephaestus is a system that can scale and generate process models for arbitrary multi-chain use cases, filling a gap in the existing literature. Hephaestus can be applied over established blockchain interoperability protocols, serving as a monitoring and audit layer, providing better response capacity and thus security. Hephaestus is implemented as a business logic plugin for Hyperledger Cactus. It can listen to local transactions emitted from multiple blockchains and derive *ccmodel*s representing arbitrary cross-chain use cases. A set of metrics and monitoring tools can be built on *ccmodel*s, allowing for a fine-grain audit of the protocol in question.

Our evaluation includes creating a cross-chain use case composed of a pair of smart contracts and cross-chain logic and executing several experiments to test the performance and reliability of `Hephaestus`. We conclude that we have low latency in generating *ccmodel*s for the given use case and that our tool can scale with the number of blockchains and *cctx*s.

We pave the way to enable a better user experience for the end user and protocol operators by enabling the analysis, monitoring, and optimization of *ccmodel*s. Use cases such as reconfiguring wallets across chains, uniformizing user interface designs across chains, managing additional base layer tokens for gas, doing tax reports, and analyzing cross-chain maximal extractable value do not need to be complicated.

Acknowledgments

This project was partially supported by *The Linux Foundation* as part of the *Hyperledger Summer Internships* program under the *Visualization and Analysis of Cross-chain Transactions* project. We thank Kevin Liao, Iulia Mihaiu, Sabrina Scuri, and Nuno Nunes for suggestions that improved this paper. This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID) and 2020.06837.BD, and by the European Commission through the contract 952226 (BIG). Rafael was supported by Blockdaemon.

References

[1] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A Survey on Blockchain Interoperability: Past, Present, and Future Trends," *ACM Computing Surveys*, vol. 54, no. 8, pp. 1–41, May 2021. [Online]. Available: http://arxiv.org/abs/2005.14282

[2] R. Belchior, L. Torres, J. Pfannschmid, A. Vasconcelos, and M. Correia, "Is My Perspective Better Than Yours? Blockchain Interoperability with Views." TechRxiv, Jun. 2022. [Online]. Available: https://www.techrxiv.org/articles/preprint/Is_My_Perspective_Better_Than_Yours_Blockchain_Interoperability_with_Views/20025857/1

[3] B. Pillai, K. Biswas, Z. Hóu, and V. Muthukkumarasamy, "Cross-blockchain technology: integration framework and security assumptions," *IEEE Access*, 2022, publisher: IEEE.

[4] P. Robinson, "Survey of crosschain communications protocols," *Computer Networks*, vol. 200, p. 108488, 2021, publisher: Elsevier.

[5] H. Qureshi, "Axelar, Bridges, and Blockchain Globalization," Jun. 2022. [Online]. Available: https://medium.com/dragonfly-research/axelar-bridges-and-blockchain-globalization-11ef3bbce9f1

[6] D. Berenzon, "Blockchain Bridges," Sep. 2021. [Online]. Available: https://medium.com/1kxnetwork/blockchain-bridges-5db6afac44f8

[7] D. Avrilionis and T. Hardjono, "Towards Blockchain-enabled Open Architectures for Scalable Digital Asset Platforms," Oct. 2021, publisher: ArXiv. [Online]. Available: https://www.scienceopen.com/document?vid=c60d84b9-911e-45a5-ab92-864ee24ec771

[8] R. Belchior, A. Vasconcelos, M. Correia, and T. Hardjono, "HERMES: Fault-Tolerant Middleware for Blockchain Interoperability," *Future Generation Computer Systems*, Mar. 2021.

[9] P. KidBold, "The Wormhole Bridge Attack Explained," Feb. 2022. [Online]. Available: https://kaicho.substack.com/p/the-wormhole-bridge-attack-explained

[10] C. Faife, "Wormhole cryptocurrency platform hacked for $325 million after error on GitHub," Feb. 2022. [Online]. Available: https://www.theverge.com/2022/2/3/22916111/wormhole-hack-github-error-325-million-theft-ethereum-solana

[11] Rekt, "Rekt - THORChain," 2022. [Online]. Available: https://www.rekt.news/

[12] R. Behnke, "Explained: The Wormhole Hack (February 2022)," Feb. 2022. [Online]. Available: https://halborn.com/explained-the-wormhole-hack-february-2022/

[13] FreddieChopin, "FYI, the hacker who exploited Harmony bridge for 100 M$ 3 days ago has already started sending stolen ETH to Tornado Cash mixer," Jun. 2022. [Online]. Available: www.reddit.com/r/CryptoCurrency/comments/vlt4xs/fyi_the_hacker_who_exploited_harmony_bridge_for/

[14] M. Barrett, "Harmony's Horizon Bridge Hack," Jun. 2022. [Online]. Available: https://medium.com/harmony-one/harmonys-horizon-bridge-hack-1e8d283b6d66

[15] C. Faife, "Nomad crypto bridge loses $200 million in "chaotic" hack," Aug. 2022. [Online]. Available: https://www.theverge.com/2022/8/2/23288785/nomad-bridge-200-million-chaotic-hack-smart-contract-cryptocurrency

[16] The Block Research, "Largest DeFi exploits," 2022. [Online]. Available: https://www.theblock.co/data/decentralized-finance/exploits/largest-defi-exploits

[17] The Straits Times, "Cryptocurrency-bridge hacks top $1.36 billion in little over a year," *The Straits Times*, Apr. 2022. [Online]. Available: https://www.straitstimes.com/tech/tech-news/cryptocurrency-bridge-hacks-top-136-billion-in-little-over-a-year

[18] N. Team, "The Road to Recovery," Aug. 2022. [Online]. Available: https://medium.com/nomad-xyz-blog/the-road-to-recovery-6abe5eec8ff1

[19] V. Buterin, "Vitalik Buterin on cross-chain bridges," 2022. [Online]. Available: www.reddit.com/r/ethereum/comments/rwojtk/ama_we_are_the_efs_research_team_pt_7_07_january/hrngyk8/

[20] H. Montgomery, H. Borne-Pons, J. Hamilton, M. Bowman, P. Somogyvari, S. Fujimoto, T. Takeuchi, T. Kuhrt, and R. Belchior, "Hyperledger Cactus Whitepaper," Hyperledger Foundation, Tech. Rep., 2020. [Online]. Available: https://github.com/hyperledger/cactus/blob/master/docs/whitepaper/whitepaper.md

[21] Dune, "Ethereum bridges TVL over time," 2022. [Online]. Available: https://dune.com/queries/118245

[22] R. Belchior, S. Guerreiro, A. Vasconcelos, and M. Correia, "A survey on business process view integration: past, present and future applications to blockchain," *Business Process Management Journal*, vol. ahead-of-print, no. ahead-of-print, Jan. 2022. [Online]. Available: https://doi.org/10.1108/BPMJ-11-2020-0529

[23] W. Van Der Aalst, "Process mining: Overview and opportunities," *ACM Transactions on Management Information Systems (TMIS)*, vol. 3, no. 2, pp. 1–17, 2012, publisher: ACM New York, NY, USA.

[24] J. Küster, K. Ryndina, and H. Gall, "Generation of business process models for object life cycle compliance," in *International Conference on Business Process Management*, vol. 4714 LNCS. Springer, Berlin, 2007, pp. 165–181.

[25] R. M. Dijkman, M. Dumas, and C. Ouyang, "Semantics and analysis of business process models in BPMN," *Information and Software Technology*, vol. 50, no. 12, pp. 1281–1294, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584908000323

[26] R. Belchior, L. Riley, T. Hardjono, A. Vasconcelos, and M. Correia, "Do You Need a Distributed Ledger Technology Interoperability Solution?" *Techrxiv 18786527/1*, Feb. 2022, publisher: TechRxiv. [Online]. Available: https://www.techrxiv.org/articles/preprint/Do_You_Need_a_Distributed_Ledger_Technology_Interoperability_Solution_/18786527/1

[27] J. Han, H. E, G. Le, and J. Du, "Survey on NoSQL database," in *2011 6th International Conference on Pervasive Computing and Applications*, 2011, pp. 363–366.

[28] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears, "Dedalus: Datalog in Time and Space," in *Datalog Reloaded*, ser. Lecture Notes in Computer Science, O. de Moor, G. Gottlob, T. Furche, and A. Sellers, Eds. Berlin, Heidelberg: Springer, 2011, pp. 262–281.

[29] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou, "Datalog and Recursive Query Processing," *Foundations and Trends® in Databases*, vol. 5, no. 2, pp. 105–195, Nov. 2013, publisher: Now Publishers, Inc. [Online]. Available: https://www.nowpublishers.com/article/Details/DBS-017

[30] L. Clarke, "How do we solve bitcoin's carbon problem?" *The Observer*, Jan. 2022. [Online]. Available: https://www.theguardian.com/technology/2022/jan/30/how-do-we-solve-bitcoins-carbon-problem

[31] T. Q. Tezos, "Proof of Work vs. Proof of Stake: the Ecological Footprint," Mar. 2021. [Online]. Available: https://medium.com/tqtezos/proof-of-work-vs-proof-of-stake-the-ecological-footprint-c58029faee44

[32] O. US EPA, "Greenhouse Gases Equivalencies Calculator - Calculations and References,"

Aug. 2015. [Online]. Available: https://www.epa.gov/energy/greenhouse-gases-equivalencies-calculator-calculations-and-references

[33] I. Mihaiu, R. Belchior, S. Scuri, and N. Nunes, "A Framework to Evaluate Blockchain Interoperability Solutions," TechRxiv, Tech. Rep., Dec. 2021. [Online]. Available: https://www.techrxiv.org/articles/preprint/A_Framework_to_Evaluate_Blockchain_Interoperability_Solutions/17093039

[34] B. Putz and G. Pernul, "Detecting Blockchain Security Threats," in *2020 IEEE International Conference on Blockchain (Blockchain)*, Nov. 2020, pp. 313–320.

[35] A. Berti, S. J. Van Zelst, and W. van der Aalst, "Process mining for python (PM4Py): bridging the gap between process-and data science," *arXiv preprint arXiv:1905.06169*, 2019.

[36] W. M. van der Aalst and A. Berti, "Discovering object-centric Petri nets," *Fundamenta informaticae*, vol. 175, no. 1-4, pp. 1–40, 2020, publisher: IOS Press.

[37] Chainlink, "Cross-Chain Interoperability Protocol (CCIP) | Chainlink," 2022. [Online]. Available: https://chain.link/cross-chain

[38] J. Zhang, J. Gao, Y. Li, Z. Chen, Z. Guan, and Z. Chen, "Xscope: Hunting for Cross-Chain Bridge Attacks," Aug. 2022, arXiv:2208.07119 [cs]. [Online]. Available: http://arxiv.org/abs/2208.07119

[39] M. Hargreaves, T. Hardjono, and R. Belchior, "Open Digital Asset Protocol draft 02," Internet Engineering Task Force, Tech. Rep., 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-hargreaves-odap-02

[40] R. Belchior, M. Correia, and T. Hardjono, "Gateway Crash Recovery Mechanism draft v1," IETF, Tech. Rep., 2021. [Online]. Available: https://datatracker.ietf.org/doc/draft-belchior-gateway-recovery/

[41] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. Knottenbelt, "Xclaim: Trustless, interoperable, cryptocurrency-backed assets," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 193–210.

[42] "Dune." [Online]. Available: https://dune.com/home

[43] Chainanalysis, "The Blockchain Data Platform - Chainalysis," 2022. [Online]. Available: https://www.chainalysis.com/

[44] Metla, "Metla - the ultimate crypto dashboard," 2022. [Online]. Available: https://metla.com/

[45] Moralis, "Moralis The Web3 Development Workflow," 2022. [Online]. Available: https://moralis.io/

[46] Rokti, "Rokti portfolio tracker," 2022. [Online]. Available: https://rotki.com

[47] Certik, "CertiK Blockchain Security Leaderboard," 2022. [Online]. Available: https://www.certik.com

[48] Token Flow, "Token Flow Insights," 2022. [Online]. Available: https://tokenflow.live

[49] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[50] B. Putz, F. Böhm, and G. Pernul, "HyperSec: Visual Analytics for blockchain security monitoring," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2021, pp. 165–180.

[51] P. Zheng, Z. Zheng, X. Luo, X. Chen, and X. Liu, "A Detailed and Real-Time Performance Monitoring Framework for Blockchain Systems," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, May 2018, pp. 134–143.

[52] M. Bartoletti, S. Lande, L. Pompianu, and A. Bracciali, "A general framework for blockchain analytics," in *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2017, pp. 1–6.

[53] D. N. Dillenberger, P. Novotny, Q. Zhang, P. Jayachandran, H. Gupta, S. Hans, D. Verma, S. Chakraborty, J. Thomas, M. Walli, and others, "Blockchain analytics and artificial intelligence," *IBM Journal of Research and Development*, vol. 63, no. 2/3, pp. 5–1, 2019, publisher: IBM.

[54] N. Tovanich, N. Soulié, N. Heulot, and P. Isenberg, "An Empirical Analysis of Pool Hopping Behavior in the Bitcoin Blockchain," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, May 2021, pp. 1–9.

[55] B. Nasrulin, M. Muzammal, and Q. Qu, "Chainmob: Mobility analytics on blockchain," in *2018 19th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 2018, pp. 292–293.

[56] R. Hobeck, C. Klinkmüller, H. Bandara, I. Weber, and W. M. van der Aalst, "Process mining on blockchain data: a case study of Augur," in *International conference on business process management*. Springer, 2021, pp. 306–323.

[57] M. Müller and P. Ruppel, "Process Mining for Decentralized Applications," in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, Apr. 2019, pp. 164–169.

[58] C. Klinkmüller, A. Ponomarev, A. B. Tran, I. Weber, and W. van der Aalst, "Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications," in *Business Process Management: Blockchain and Central and Eastern Europe Forum*, ser. Lecture Notes in Business Information Processing, C. Di Ciccio, R. Gabryelczyk, L. García-Bañuelos, T. Hernaus, R. Hull, M. Indihar Štemberger, A. Kő, and M. Staples, Eds. Cham: Springer International Publishing, 2019, pp. 71–86.

[59] R. Mühlberger, S. Bachhofner, C. Di Ciccio, L. García-Bañuelos, and O. López-Pintado, "Extracting Event Logs for Process Mining from Data Stored on the Blockchain," in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing, C. Di Francescomarino, R. Dijkman, and U. Zdun, Eds. Cham: Springer International Publishing, 2019, pp. 690–703.

[60] F. Corradini, F. Marcantoni, A. Morichetta, A. Polini, B. Re, and M. Sampaolo, "Enabling Auditing of Smart Contracts Through Process Mining," in *From Software Engineering to Formal Methods and Tools, and Back: Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*, ser. Lecture Notes in Computer Science, M. H. ter Beek, A. Fantechi, and L. Semini, Eds. Cham: Springer International Publishing, 2019, pp. 467–480. [Online]. Available: https://doi.org/10.1007/978-3-030-30985-5_27