Data Protection in Permissioned Blockchains using Privilege Separation

Arun Joseph, Nikita Yadav, Vinod Ganapathy
Indian Institute of Science Bangalore, India

Dushyant Behl, Praveen Jayachandran
IBM Research, India

2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS) | 978-1-6654-7706-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/COMSNETS56262.2023.10041304

***ABSTRACT*—This paper concerns the Hyperledger Fabric permissioned blockchain system. This system is in popular use in several enterprise settings, where each participating corporate entity may have sensitive business-related data whose confidentiality it wishes to protect. Fabric provides the channel abstraction that ensures that channel data (*e.g.,* data stored in that channel's ledger, or data transmitted via the network to members of that channel) are only accessible to members of that channel. Unfortunately, as we show in this paper, the channel abstraction only offers data protection under the implicit assumption that all system components in the permissioned blockchain are trustworthy. This assumption may not hold in the presence of compromised container nodes, on which several blockchain-related components execute, or malicious business users inside any one of the participating corporate entities. Under such situations, sensitive corporate data can be leaked to unauthorized entities.**

We present Aramid, which is an enhanced version of Fabric that offers data protection even in the presence of compromised blockchain components. Aramid uses a privilege-separated architecture in which blockchain components (such as peer or orderer nodes) that are members of multiple channels execute on different containers. Aramid is transparent to legacy Fabric applications, requiring no changes to their codebase. Through our prototype implementation, we show that Aramid robustly defends against a number of attacks possible on Fabric, and that it does so with performance comparable to Fabric.

***Index Terms*—permissioned blockchain, privilege separation, security, data leakage**

I. Introduction

A number of domains have now adopted blockchain-based technologies for a variety of applications. While *public* blockchains have received much attention in applications such as cryptocurrencies, *permissioned* blockchains are increasingly being deployed in a number of enterprise applications. The participants in a public blockchain are generally pseudo-anonymous Internet-based entities, and the main goal is typically to ensure the integrity of transactions and achieve consensus. In contrast, the identity of the participants in a permissioned blockchain is known to other authorized participants. For example, a permissioned blockchain may be used by the organizations that participate in a supply chain [1], trade or finance [2], [3], and those offering digital identities [4]. In such networks, each participant must be explicitly authorized by the network consortium members, and identities of the system components contributing to the ledger and maintaining a copy of the ledger (called *peer* nodes) are known to all the blockchain participants. Given the inherent confidential nature of business transactions, permissioned blockchains not only ensure integrity of transactions, but also aim to protect data access through identity verification, authorization and access control mechanisms.

Blockchains ensure integrity of transactions by maintaining an append-only, immutable, hash-chained ledger of transactions managed through decentralized consensus among a set of peer nodes. They guarantee integrity by tolerating a fraction of peer nodes being faulty or malicious based on the type of consensus protocol used. There is a large body of work on consensus algorithms providing different integrity guarantees [5]–[8] and also work on verification and finding bugs in smart contracts [9]–[11] to ensure integrity. However, even a single misbehaving peer node (*e.g.,* due to compromise or an insider attack) could compromise data protection by leaking blockchain data to an attacker. Blockchains provide no protection against such attacks and they remain an unaddressed problem. *The central contribution of this paper is providing data protection on the blockchain even when peer processes and other blockchain components (such as smart contracts) are compromised.*

Blockchain data protection has primarily been addressed through application-layer mechanisms [12], [13], such as encrypting the data at the application before storing them on the blockchain. This, however, negates the power of smart contracts to perform decentralized computation on the blockchain data. Techniques such as zero-knowledge proofs have been leveraged to provide additional application integrity guarantees while providing data confidentiality [14], such as proof against double spending or proof of solvency [15], *i.e.,* asset balances are non-negative at all points in time. Secure multi-party computation has been proposed for use with blockchains for more general purpose computation on private data [16]. These techniques still do not protect against a malicious peer revealing transaction logs or membership information, while also causing significant overhead in performance and application complexity.

The focus of this paper is on Hyperledger Fabric [17], a popular permissioned blockchain platform, although the proposed ideas can be extended to other permissioned blockchain platforms as well with some effort. In Fabric, peer nodes that manage the distributed ledger could belong to different organizations. Peers can run smart contracts (called *chaincodes* in Fabric) that implement business logic and operate on the ledger data. Each peer can be part of one or more *channels*.

Each channel is associated with a ledger and an independent consensus (or *ordering service*) to determine the order of transactions and blocks to be added to the ledger. The ledger stores transactions carried out within the channel. Fabric also supports *private data collections* that allow certain data in a channel to only be shared among a subset of organizations. Organizations that are part of the private data collection may endorse, commit, and query the private data through smart contracts. Unfortunately, Fabric (and similarly other permissioned blockchain implementations) provides a number of avenues by which confidentiality of channel data can be violated. Peers and other Fabric components run as containers. Leaks of confidential ledger data and cross-channel data leaks can happen whether through exploitable vulnerabilities in the containers, Fabric components, and smart contracts, or adversarial insiders seeking sabotage. We present a number of examples of such attacks in Section IV.

In this paper, we propose ***Aramid***, an enhanced version of Fabric that aims to improve data protection. The key problem in Fabric is that it offers too large an attack surface—implementing data protection atop Fabric would require placing trust on too many system components and business users. In Aramid, peers and other untrusted components that are part of multiple channels are partitioned to run as different instances, with a separate instance per channel. It ensures that data belonging to a channel is only accessible from system components and business users that have privileges to access that channel. Unlike Fabric, where blockchain components (*e.g.,* peers, orderers, smart contracts) must be trusted in order to achieve data protection, in Aramid, these are untrusted. Instead, Aramid shifts the trust to vastly simpler and smaller *trusted proxies*, as discussed below.

Partitioning peers into different instances by channel improves data protection, but breaks the existing Fabric abstraction of a single representative per enterprise (*e.g.,* in cases where a single peer from an enterprise participates in multiple channels). Aramid thus strives to preserve the existing abstractions in Fabric in an attempt to ensure that applications do not have to be rewritten. Aramid thus introduces trusted proxies that serve as the front-end to these privilege-separated peer instances. A client application interacting with an enterprise interacts transparently with the trusted proxy for that enterprise, which in turn routes the information to the suitable instance, based on channel information. Further, we use network policies to restrict communication of each Fabric component. For instance, each peer is permitted to communicate only with its trusted proxy and a smart contract process is only permitted to communicate with the peer that initiated it. All other attempted communication is dropped by the host network layer, protecting the system against malicious or sabotaged user (peer or smart contract) processes.

Aramid's privilege-separated architecture yields a vastly reduced attack surface in comparison to Fabric. In particular, peers and orderers are no longer part of Aramid's TCB (they are in Fabric), and are instead replaced by the significantly smaller and logically-simpler trusted proxy. The trusted proxy in Aramid is under 5% (by LOC of Go code) of the size of peers and orderers in Fabric (see Figure 2).

We have implemented Aramid for Fabric v2.2 and have deployed it atop the Kubernetes [18] container orchestration framework. Aramid uses the Istio service mesh [19] to enforce network-level policies for data protection. In our evaluation, we show that Aramid robustly defends against several data protection-violating attacks that are possible on Fabric, and that it does so with performance comparable to Fabric. In summary, this paper's contributions are as follows:

- **Identifying data protection failures in Fabric.** We argue that Fabric's data protection model requires implicitly trusting all system components in the permissioned blockchain. We present a number of attacks that are possible if this assumption is violated, *e.g.,* through buggy/compromised containers or insider attacks (Section IV).
- **Design and implementation of Aramid.** We present Aramid's privilege-separated architecture that improves data protection in the presence of compromised blockchain components (Section V).
- **Evaluation.** We present a security and performance evaluation that shows that the performance (throughput and latency) achieved at client applications that use Aramid is comparable to that of Fabric (Section VI).

II. Background on Fabric

In this section, we provide background on Hyperledger Fabric [17] and its assumed threat model. We describe the different participant roles in Fabric along with some of the design constructs available to support data protection.

Peers are owned and operated by organizations and are the organization's connection points to the network. Each peer maintains a copy of the blockchain ledger and is responsible for its integrity. They execute business logic deployed as smart contracts (*a.k.a.* chaincodes) on them and each successful invocation of a smart contract is appended as a transaction to the blockchain ledger. A peer process may connect to several other components over the network—client applications, ledger, smart contract processes, ordering service nodes, and other peer nodes.

Channels are private sub-networks among member organizations (with peers), each having its own shared ledger, smart contracts, and ordering service nodes. A channel is a logical entity, which allows confidential communication and transactions among its members. Organizations that are not part of a channel should not have information about that channel. A single peer process belonging to an organization could participate in one or more channels and could thus maintain the ledgers of multiple channels using a single database instance.

In any organization participating in a blockchain platform, there are typically two personas of interest. The first is a *business user*, whose application data is recorded on the ledger through transactions performed on the blockchain. They understand the value of the data and any confidentiality

and protection needs associated with it. They may also be subject to compliance and audit of business and regulatory policies concerning data protection. The second is an *IT system administrator* who provides the infrastructure for hosting the blockchain, including the peer, ledger, smart contract processes and ordering service nodes. They are responsible for the availability and IT security of the infrastructure, but any data stored on the blockchain is only a string of bytes to them.

Fabric's peer and smart contract processes can be compromised and leak data to other processes or send unencrypted confidential data, which can be captured by an attacker. Further, a compromised peer process can leak secret channel-specific blockchain data and information to another Fabric process that is not part of that channel. Similarly, a compromised ordering service node can send blocks of a particular channel to peers who are not part of that channel.

III. Threat Model for Aramid

Aramid aims to prevent attacks by compromised or malicious Fabric system components that violate data protection. Aramid relieves blockchain business users who control all Fabric processes and the ledger in an organization from the obligation of data protection. We consider that *all Fabric processes and business users that can access them are untrusted*, and can be compromised and act maliciously. Aramid provides systemic means for enforcing data protection rules at the infrastructure layer, following the principle of privilege separation. This architecture offers a vastly reduced attack surface in comparison to Fabric.

On average a single IT administrator may manage tens or even hundreds of applications (and therefore hundreds or more business users using those applications). Further, they do not comprehend the value of the bytes stored to maliciously leak them. By *trusting only the infrastructure layer and few IT administrators who manage them*, Aramid also reduces the number of insiders trusted for data protection. Specifically, in the infrastructure layer, we trust the operating system for not modifying and leaking packet information. We run all processes in a Kubernetes [18]-based infrastructure, and we trust the Kubernetes control plane for providing pod-level isolation (a *pod* is a group of one or more containers in Kubernetes), network, and storage security.

A malicious insider (business user) on Fabric can access all data from all channels in which the business participates. While Aramid reduces the attack surface, it does not eliminate data exposure. For instance, a malicious business insider with credentials to access a particular channel's data can still leak that channel's data via out-of-band methods (*i.e.,* out of the purview of the blockchain). Future extensions could combine Aramid with prior art (*e.g.,* [20]) to mitigate such attacks.

IV. Motivating Examples

To motivate examples of data protection violations in Fabric, we consider an example business scenario and present several attacks that are possible with Fabric. We emphasize that these attacks are merely illustrative, and not in any way a comprehensive listing of data protection attacks that Aramid defends against.

Consider a Fabric deployment that is used by shipping companies and delivery companies (*e.g.,* TradeLens [1]). Shipping companies publish a list of available container slots on their ships. Delivery companies bid for these container slots, with each delivery company submitting a bid that consists of the number of slots it is bidding for, and the price offered per slot. Each shipping company then allocates container slots based on a variety of business considerations, such as the price offered per slot, long-term relationship with the delivery company, its reputation, and prior business history with that company.

Each shipping company's algorithm to select bid winners is proprietary, and the shipping company may also wish to keep secret the list of bids that it receives. Similarly, each delivery company will want to protect the confidentiality of the price that it quotes and the number of slots that it requests. For example, a Fabric deployment could contain:

- A single public channel, available to all shipping and delivery companies, to share common data among all participants. This channel's ledger records the availability of container slots, shipping data, and other common data.
- One channel per pair of shipping and delivery companies that wish to do business with each other. The channel's ledger records data private to that pair of companies, *e.g.,* the bid by the delivery company to that shipping company.
- One channel per shipping company and consortium of delivery companies that wish to collaborate in submitting a bid. For example, an individual delivery company may not have sufficiently many goods to fill a single container, in which case it may choose to share (and submit a bid for) a container together with collaborating delivery companies. The ledger of this channel records joint bids submitted by the consortium of delivery companies.

The permission infrastructure of Fabric ensures that only channel members can access the ledger of that channel. In fact, Fabric's infrastructure even protects channel-membership information. Thus, for instance, it would not be possible in the above example for a business user in one delivery company to determine if its competitor is in business with a particular shipping company. As another example, one member of a consortium of delivery companies cannot determine if another member is part of a second consortium.

While these security checks are embedded in Fabric's permission-checking mechanism, they offer data protection only under the implicit assumption that all components of the blockchain deployment are well-behaved. Unfortunately, as multiple CVEs and bug-reports show (*e.g.,* [21]–[31]), this assumption can easily be broken by compromised blockchain components (*e.g.,* peers, orderers) and malicious business users.

We present below several examples of attacks that violate data protection in the above setting:

① **Sending data to entities outside the blockchain.** Fabric's permission infrastructure protects a channel's ledger data by

ensuring that only peers with channel membership can access it. However, Fabric does not have in-built mechanisms that prevent a malicious peer from leaking ledger data to entities outside the blockchain.

② **Sending unencrypted data.** A compromised or malicious peer can send data unencrypted. A network-based adversary snooping on network links can read all the data sent by the peer. This may include sensitive data, transaction inputs and outputs processed by smart contracts that the peer executes. This attack has the same effect as that of sending data to entities outside the blockchain.

③ **Intra-container data leaks.** Fabric components run within Docker containers on a Kubernetes cluster. In a maliciously configured container, a Fabric-related process (*e.g.,* one running the code of a peer) could leak data to other processes within that container, which in turn may send data to external unauthorized entities.

④ **Inter-channel data leaks.** Fabric relies on channels as the fundamental data partitioning abstraction. However, a malicious Fabric component can leak data that is accessible.

⑤ **Leaking channel membership information.** In Fabric, even membership in a channel is protected information that is privy only to members of that channel. In the example above, a compromised peer belonging to a delivery company that is part of multiple consortia can leak information about the members of one consortium, either to other consortia or to external entities.

⑥ **Leaking smart contract business logic information.** The business logic within smart contracts (*e.g.,* the proprietary algorithm used by a shipping company for bid selection) is considered confidential. The code of a smart contract executing in one channel is only known to members of the channel, in fact, only known to the endorsers for that smart contract. A malicious system component process could leak business logic information to outside the channel.

⑦ **Leaks via non-channel messages.** In Fabric, peer nodes send out messages that are not tied to any individual channels. A malicious peer can leak channel data to an adversary by encoding sensitive information as part of these non-channel messages.

⑧ **Leaking data stored in private data collections.** Fabric supports the notion of private data collections. It allows a subset of channel members to form a private collection and transact privately. A malicious peer can easily leak private data accessible to it to a non-collection member.

To summarize, Fabric provides the channel abstraction for data protection. The mechanisms to enforce this protection are part of Fabric's permission model, but Fabric assumes that the various system components are not malicious. In the presence of malicious components, such as peers, smart contracts and ordering services, the assumptions that Fabric makes no longer hold, leading to the kinds of attacks discussed above. Aramid aims to improve Fabric by providing data protection even in the presence of malicious components.

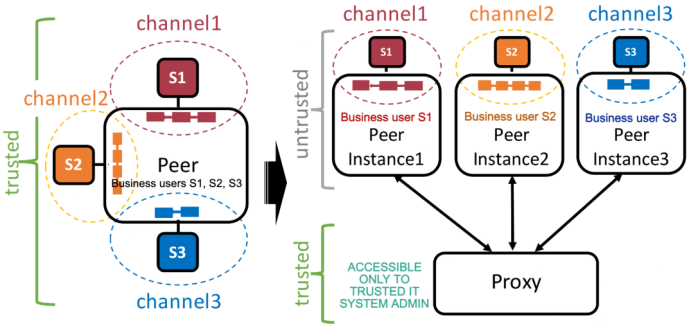

Fig. 1. This figure compares how Fabric and Aramid organize peers. A single peer instance that is part of multiple channels in Fabric is replaced by multiple peer instances (one per channel) in Aramid. Each peer can run smart contracts (denoted by "S1", "S2", "S3") and can only access the ledger of that channel. A trusted proxy acts as a front for the multiple peer instances to external applications. The proxy inspects messages directed to the peers, and routes the messages appropriately, based on the channel. In Aramid, peers are untrusted, and the trust is instead shifted to a proxy node.

V. ARAMID

Aramid extends Fabric's data security guarantees even to situations when Fabric system components are compromised. We designed Aramid with three objectives in mind:

- **Reduce the number and complexity of trusted components:** As already discussed, Fabric's channel-based data protection mechanisms only work as expected when each component (peers, orderers, smart contracts) behave honestly. In a large blockchain deployment this assumption may no longer hold. Aramid aims to offer data protection while trusting far fewer and logically-simpler components than on Fabric.
- **Multi-faceted policy enforcement:** Aramid enforces policies both at the network layer, ensuring messages are visible only to (authorized) participants in the blockchain, as well as Fabric-specific policies. For instance, the notion of channels is unique to Fabric, and Aramid ensures that channel messages are visible only to other nodes that are part of the channel.
- **Preserve Fabric components and abstractions:** Aramid is transparent to legacy Fabric applications. Existing applications do not have to reprogrammed. Thus, any interaction with a peer node in Fabric will continue in the same way in Aramid. Aramid transparently introduces new, trusted network elements that enforce policies without breaking the illusion of communicating directly with a peer node. Moreover, Aramid can operate and provide a measure of protection even if its mechanisms are adopted by only *some* participating entities of the blockchain network.

A. Enforcing Channel Data Separation

Fabric's design allows a peer node (*i.e.,* a process running the peer code in a container instance) to be part of multiple channels, and consequently, access the ledgers that are private to these channels. As shown in Section IV, a compromised or malicious peer process can leak sensitive channel data to any network member who does not have explicit access to that channel's data.

Aramid's core design is based on privilege separation that

replaces a single peer node with multiple peer nodes such that each peer is part of only one channel and maintains only one ledger (illustrated in Figure 1). Each such peer node runs in a Docker container inside a fresh pod instance. Each peer node has its own smart contract instances for its channel as separate pods that it alone communicates with.

Simply disaggregating a single peer node in Fabric into multiple peers breaks transparency to applications, *e.g.,* other Fabric nodes and client nodes that interact with the peer to perform blockchain transactions. These applications will now have to route messages to different peer instances, based upon the channel on which the corresponding messages are being exchanged. Unfortunately, this will break existing applications. To maintain transparency, Aramid instead introduces a *trusted proxy node* for the peer instances. Applications that interacted with the single peer in Fabric interact instead with this trusted proxy in Aramid, which then routes messages to the appropriate peer instance based on channel information. The proxy node is part of Aramid's TCB and transparently mediates all communications to and from these peer instances. The collection of proxy and peer instances represent a single "peer" entity from the perspective of the Fabric network as well as any client applications interacting with it.

Aramid enforces two kinds of policies:

① *Fabric-specific policies* that are enforced by trusted proxies. The most notable of these policies is that two communicating Fabric components that exchange channel-specific messages must belong to the same channel. As we will describe, the trusted proxy has the privilege to inspect all incoming messages to and outgoing messages from the peers for which it serves as a proxy. If the above policy is violated, then the proxy silently discards the message. Although we have illustrated proxies so far only for peer nodes, such trusted proxies can be added for every Fabric component.

② *Generic network policies* that apply to any permissioned blockchain deployment. These policies are enforced by the underlying service mesh and container orchestration framework (Istio and Kubernetes, in our implementation). The trusted proxy interacts with the service mesh and dynamically adds these network policies that restrict communication paths.

At first blush, it may appear that Aramid's design simply shifts the trust boundary from peer and orderer nodes to the trusted proxy nodes. While this is indeed the case, there are two key benefits to this approach:

① System components such as peers and orderers are complex pieces of code that have evolved over time, and contain complex logic to handle the plethora of messages and corner cases that Fabric supports. In contrast, the trusted proxy is extremely simple in its functionality, and only serves to route messages to the correct peer or orderer node, based on channel information. Because messages on the Fabric network are encrypted, the trusted proxy also manages TLS keys on behalf of the peers and orderers so that it can inspect the packets for channel information. The proxy implements no other functionality besides routing, key-management, and packet header inspection to glean channel information.

Component	LOC (Go code)	# Vendor modules used
Peer	1,41,031	18
Orderer	55,501	16
Trusted proxy	8,541	13

Fig. 2. Sizes of various components in Fabric (shaded gray) and Aramid. Peers and orderers are part of Fabric's TCB. The trusted proxy is part of Aramid's TCB, but peers and orderers are not.

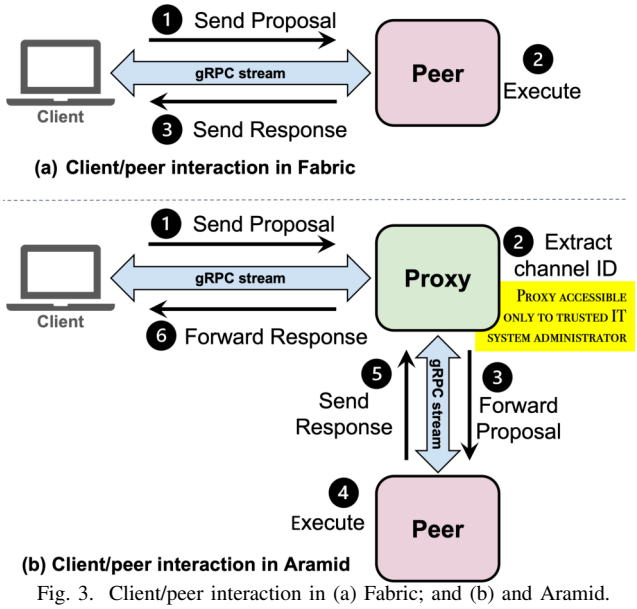

Fig. 3. Client/peer interaction in (a) Fabric; and (b) and Aramid.

② The trusted proxy is managed by IT system administrators, and is not accessible to business users. As discussed in Section II, Aramid does not trust business users, who are cognizant of the value of the ledger data.

As a result of this simplicity, the attack surface of the trusted proxy in Aramid is vastly smaller than that of peer nodes and orderer nodes, which are part of the TCB in Fabric. We quantify this reduction in Figure 2, which shows the sizes (in lines of Go code) of the components that are part of the TCB in Fabric versus the size of the trusted component (the core of Aramid's TCB).

B. Implementation

We have implemented a prototype of Aramid for Fabric v2.2 and deployed it on Kubernetes [18], a popular container management platform for deploying and managing Fabric networks. Each Fabric component, including peer instances, smart contracts, ledger, orderer, and the trusted proxy, runs in a separate Kubernetes pod. We leverage the open-source service mesh Istio [19] to ensure that only encrypted messages are exchanged on the network.

We next describe in detail how different transaction flows and design aspects of Fabric are transparently supported to client applications in Aramid.

Transaction submission: Figure 3(a) illustrates the interaction between a client and a peer in Fabric for transaction submission. It involves the following steps:

① *Send proposal.* The client initiates the transaction by creat-

ing a transaction proposal that includes the smart contract ID, function name, arguments, channel ID, and client certificate, among others. The Fabric SDK suitably formats and packages the transaction proposal (as a protocol buffer over gRPC) and takes the client's cryptographic credentials to produce a unique signature for this transaction proposal. This transaction proposal is then sent to one or more endorsing peers.

② *Execute transaction.* The peer performs verification checks on the transaction proposal. It then simulates the proposal by executing the operation on the specified smart contract.

③ *Send response.* The endorsing peer replies with the proposal response and its signature over the proposal response.

In Aramid, the client-peer interaction is mediated by a trusted proxy, as shown in Figure 3(b). It provides the illusion to the client that it is interacting directly with the peer, as illustrated in the steps below:

① *Send proposal.* The client application initiates the transaction and creates a transaction proposal. It sets up a gRPC connection with the proxy. The client, under the illusion that the proxy is the endorsing peer, makes a gRPC function call to the proxy with the signed proposal.

② *Extract channel ID.* In the gRPC function, the proxy verifies the client's identity and extracts the channel ID from the content of the transaction proposal.

③ *Forward proposal.* The proxy keeps a record of the specific channel for which each peer instance is responsible. Using this mapping information and channel ID (extracted in step 2), it finds the correct peer instance. It then makes the gRPC function call with the signed transaction proposal on behalf of the client and waits for the response.

④ *Execute transaction.* The peer instance performs verification checks on the transaction proposal. It then executes the transaction on the specified smart contract and produces the response. This step is identical to Fabric.

⑤ *Send response.* The peer instance returns the signed transaction response to the proxy. The peer instance code is identical to a Fabric peer instance (*i.e.,* unmodified).

⑥ *Forward response.* The proxy returns the gRPC function call with the signed response from the peer instance.

VI. Evaluation

We evaluate two aspects of Aramid. First, we evaluate its ability to protect data in the presence of compromised Fabric components. We illustrate how Aramid's defense mechanisms handle the example attacks listed in Section IV. Second, to evaluate the performance of Aramid and compare it against Fabric, we measure the latency and throughput of committing transactions on the blockchain.

A. Experimental Setup

We set up a Fabric-based testbed for our experiments, consisting of three organizations. There is one orderer node and each organization has a single peer node. The default parameters we used for configuring the peer and orderer nodes are shown in Figure 5.

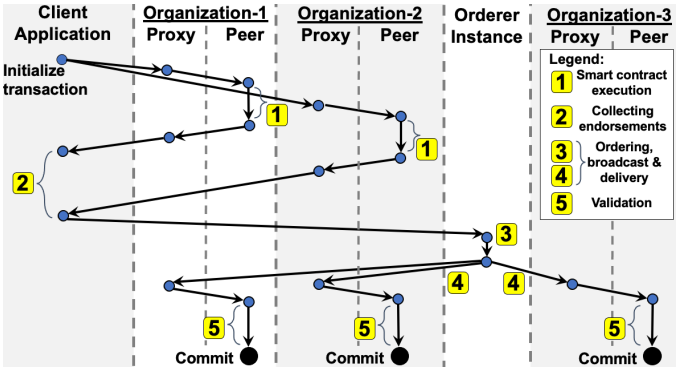

Fig. 4. Interaction between components in Aramid in fulfilling a transaction. In this example, a transaction initiated by the client executes a smart contract at Organization-1 and Organization-2, following which the endorsements are collected, and the endorsed transactions are sent to the ordering service. The orderer then broadcasts it to all the peers, who validate the transaction and add it to their ledgers.

For the performance evaluation experiments, we use Hyperledger Caliper [33] as our workload generator. We run all of our experiments with the Marbles smart contract [32]. We run each organization on a separate Kubernetes cluster node and the resource configuration is kept same for a head-to-head performance comparison of Fabric and Aramid. Our experimental network configuration matches those in real deployments, so we believe that the benchmark results are representative of a real-world deployment of Aramid.

B. Evaluating Aramid's Security Robustness

In Section IV, we listed the attacks that can be launched in Fabric. We now show how Aramid successfully defends against these attacks, focusing on the mechanism(s) that prevents each of these attacks. We first review the mechanisms in Aramid:

- **TLS by default.** Aramid uses the open-source service mesh Istio [19] to ensure that all the data leaving a pod is encrypted. Istio deploys a sidecar proxy in every pod that intercepts all communications between pods. It provides transparent TLS encryption support without requiring any application changes.
- **Trusted proxy.** The trusted proxy provides complete mediation for all communications to and from Fabric components (such as peers and orderers) and enforces Fabric-specific policies. It makes routing decisions based on message content and network information.

Parameter	Values
No of Channels	2
State Database	GoLevelDB
Endorsement Policy	Any one peer on channel endorses
Batch Size	500 transactions per block
Batch Timeout	2 sec
Validation Pool Size	64
Caliper Transaction Send Rate	1500 tps
Caliper Workers	16 (local)
Smart Contract (Chaincode)	Marbles [32]
Chaincode Transaction	initMarble [32]

Fig. 5. Default configuration used for experiments.

• **Network policies.** Kubernetes network policies are firewall rules which monitor traffic to, from, and between pods. Aramid leverages these rules to block all traffic that is not explicitly allowed. Trusted proxies in Aramid define these network rules dynamically in the form of network paths for each pod, when it is created.

We now discuss which component of Aramid prevents the various attacks listed in Section IV.

① **Sending data to entities outside the blockchain.** Aramid prevents a compromised peer belonging to a shipping company from leaking the bid values to an outside entity with *network policies and policy checks at the trusted proxy*. Aramid's network rules restrict a compromised peer to interact with only the trusted proxy pod and smart contract pods with the network rules. It cannot send any data outside the network itself.

② **Sending unencrypted data.** Aramid uses TLS by default for all network connections, and this is enforced by the underlying service mesh. This attack is not possible on Aramid—all data leaving a pod is always encrypted.

③ **Intra-container data leaks.** When a container is misconfigured, or vulnerable Fabric components in it get exploited, the corresponding process can leak confidential information accessible to the container. Aramid follows standard security best-practices for Docker and Kubernetes and runs containers unprivileged, restricting file system access to read-only, and limiting interactions with the host.

④ **Inter-channel data leaks.** When a compromised Fabric peer is part of multiple channels, it can leak information from one channel to another. Aramid runs a separate peer instance per channel, each running on a different pod, and restricted in its communication using the network rules and pod security policies, thereby preventing the attack by construction.

⑤ **Leaking channel membership information.** In Aramid, a peer (or orderer) is part of only one channel and does not know about the existence of any other channels on the network. The *trusted proxy* handles new channel membership for a peer. While a peer has access to membership information for the channel it serves, it has no means of leaking this information to external entities.

⑥ **Leaking smart contract business logic.** In Aramid, smart contracts are deployed as external contracts, running in separate pods. A compromised smart contract cannot leak the business logic to anyone else but the peer instance on which it is deployed because it is restricted to communicate only to the peer instance. The compromised peer instance is restricted to communicate only to its trusted proxy, allowing Aramid to enforces policies to verify the message destination and content.

⑦ **Leak via non-channel messages.** In Fabric, a compromised peer can cleverly craft a non-channel gossip message with channel data and send it to peers on a different channel. In Aramid, the *trusted proxy* handles the non-channel gossip messages and responds on behalf of all the peer instances to prevent such attacks.

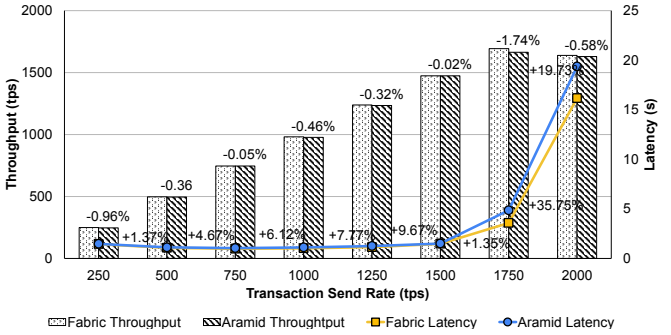

Fig. 6. Measuring the impact of transaction send rate on throughput and latency in both Fabric and Aramid.

⑧ **Leaking private data stored in private data collection.** A malicious Fabric peer can send private data to peers that are not part of that collection. Aramid's proxy prevents this by checking the gossip message content and collection policy before forwarding the private data to other organizations.

C. Performance Evaluation

In this section, we compare the performance of Aramid with that of Fabric by varying several parameters: (1) the workload send rate; and (2) the number of channels.

The major difference between Fabric and Aramid is the addition of the trusted proxy in Aramid. Our experiments are designed to analyse the performance impact of adding this new network element on the overall performance of the blockchain. The throughput and latency numbers presented in this section are averaged over 10 runs. Each run completes 200,000 transactions.

Impact of Transaction Send Rate Figure 6 plots the average throughput and latency achieved in Fabric and Aramid with different transaction send rates. Transactions are sent by Caliper workers. As our default endorsement policy is OR(Organization-1, Organization-2, Organization-3), *i.e.,* that the transaction must be endorsed by the membership service provider of any one of these organizations, Caliper is configured to send transaction proposals to only the peer of Organization-2 and get endorsements from it. Other parameters are configured as per Figure 5.

We observe that *the throughput profile of Fabric and Aramid closely match over varying transaction send rates.* With an increase in transaction-send rate the throughput increases linearly until it saturates at about 1670 transactions per second (tps) for both Fabric and Aramid.

For lower send rates (250 tps and below) the latency is little higher than that of higher send rates. This can be attributed to cases where the send rate is below block size which is set to 500 transactions. This increases latency because the system waits for the block to be filled or until the block-creation timeout. Overall, however, our conclusion from this experiment is that the performance of Aramid compares favourably with that of Fabric. On average, Aramid throughput is 0.56% less than Fabric.

Impact of Multiple Channels Figure 7 plots impact of increasing the number of channels on the average throughput and latency in Fabric and Aramid. We create six channels between

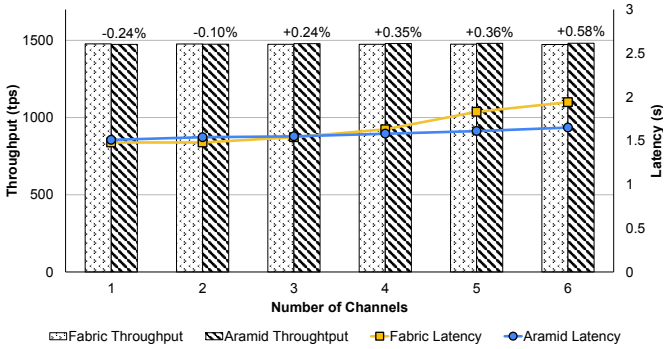

Fig. 7. Measuring the impact of the number of channels on throughput and latency in Fabric and Aramid.

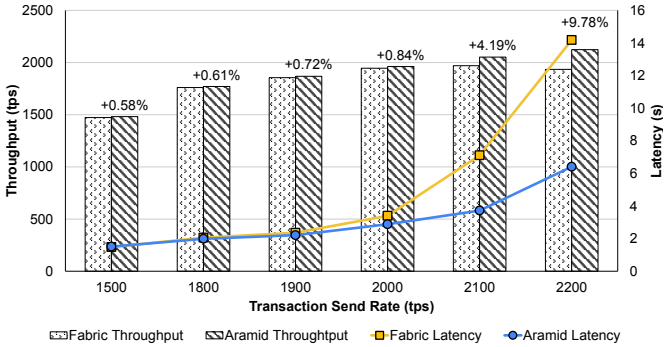

Fig. 8. Fabric and Aramid with six channels.

Organization-1 and Organization-2. We join the endorsing peer of Organization-2 on all channels. Caliper sends transactions simultaneously on multiple channels with a transaction send rate of 1500 tps evenly split among the available channels.

We first observe that *both Fabric and Aramid maintain nearly constant throughput and latency as the number of channels is increased.* As Figure 7 shows, the throughput is at around 1475tps and latency less than 2s. We note that *the throughput and latency of Aramid is slightly better than Fabric when number of channels increases.*

To study this trend in more detail, we conducted another experiment with six channels set and varied the transaction send rate. As Figure 8 shows, Fabric's throughput saturates at around 1950 tps, while Aramid's throughput saturates at 2100 tps. After the saturation point, the difference in the latencies increases. Aramid outperforms Fabric because it has a separate peer instance and separate smart contract instance for each channel. Our conclusion from this experiment is that Aramid performs better when number of channels increases (in this case, more than six channels).

VII. Related Work

Confidentiality of data for blockchain applications has in the past been supported using cryptographic mechanisms in the application layer (*e.g.,* [12]–[14], [34]). Zerocash [14], for instance, extends the Bitcoin protocol by adding new types of transactions on a separate privacy-preserving currency, where the transactions hide the sender, the recipient as well as the quantum of payment. This is achieved using zero knowledge proofs, specifically zk-SNARKs.

Recent work has leveraged secure multiparty computation (MPC) to support general-purpose computation on private data, leveraging blockchain for properties such as auditability, verifiability and fairness [35]–[38]. In both ZKP and MPC-based confidentiality mechanisms, confidential data is held privately and not stored on the blockchain and the application is written in a specific manner to provide confidentiality guarantees. Blockchain is leveraged as a tamper-proof auditable log. In contrast, Aramid aims to support confidentiality for application data stored on the blockchain even in the midst of malicious or compromised actors, and in a manner that is transparent to legacy Fabric applications. Several works have analyzed the security of container implementations, and many vulnerabilities have been found [21]–[25]. An attacker could exploit these vulnerabilities to write to arbitrary files, execute arbitrary code (with root privileges), set arbitrary Linux Security Modules (LSM), or gain access to control regions of the device attached to the host PCI bus. Lin *el al.* [39] collect an attack dataset of 223 exploits, of which 56.8% are effective against default container configurations.

Several works propose design changes to container implementations to enhance their security. The proposed methods include, for example, (a) employing proxies to mediate all commands from end-users to the docker daemon for least-privilege enforcement [40]; (b) using lightweight VMs as a faster and safer replacement for containers [41]; and (c) improved security policy enforcement using namespaces [42] and using `seccomp` [43].

Finally, we note that there is a long history of work on privilege separation in the systems security community. Privtrans [44] is an early tool that aimed to automatically transform source code into multiple, privilege-separated components, using programmer-supplied annotations. Privilege separation methods have also been investigated for specific kinds of applications, such as databases [45], HTML5 Web applications [46], [47], Web servers [48] and Android applications [49]. Diesel [45], which applies privilege separation for database access, is closest in spirit to the design of Aramid. Applications that access the database are split into multiple modules (each isolated and running with reduced privileges), with a proxy mediating access to the database. Aramid's design does not modify the source code of any Fabric components, but instead works with existing Fabric components, and applies data isolation at the granularity of channels, even in the presence of compromised containers.

VIII. Conclusions

Aramid improves upon the channel-based data protection mechanisms of Fabric. Aramid's privilege-separated design allows us to relax the assumptions that Fabric makes about all participants being trusted, and provides channel-based data protection even in the presence of compromised peers, orderers and smart contracts. Aramid is transparent to end-user applications, thereby allowing interoperability with legacy Fabric applications, and provides transaction latency and throughput rates comparable to that of Fabric.

REFERENCES

[1] Tradelens, "Tradelens: Digitizing the global supply chain," 2021.

[2] We.Trade, "We.trade," 2021. [Online]. Available: https://we-trade.com

[3] M. P. Network, "Marco polo network," 2021. [Online]. Available: https://www.marcopolo.finance/

[4] I. Blockchain, "Blockchain for digital identity," 2021. [Online]. Available: https://www.ibm.com/blockchain/solutions/identity

[5] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[6] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[7] K. Lei, Q. Zhang, L. Xu, and Z. Qi, "Reputation-based byzantine fault-tolerance for consortium blockchain," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2018, pp. 604–611.

[8] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, 2011, pp. 1–7.

[9] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *NDSS*, 2018.

[10] 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.

[11] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "{TXSPECTOR}: Uncovering attacks in ethereum from transactions," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2775–2792.

[12] E. Cecchetti, F. Zhang, Y. Ji, A. Kosba, A. Juels, and E. Shi, "Solidus: Confidential distributed ledger transactions via pvorm," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[13] A. Poelstra, A. Back, M. Friedenbach, G. Maxwell, and P. Wuille, "Confidential assets," in *International Conference on Financial Cryptography and Data Security*. Springer, 2018, pp. 43–63.

[14] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *IEEE Symposium on Security and Privacy*, 2014.

[15] G. G. Dagher, B. Bünz, J. Bonneau, J. Clark, and D. Boneh, "Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[16] F. Benhamouda, S. Halevi, and T. Halevi, "Supporting private data on hyperledger fabric with secure multiparty computation," in *2018 IEEE International Conference on Cloud Engineering, IC2E*. IEEE Computer Society, 2018.

[17] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, and et al., "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *ACM Eurosys*, 2018.

[18] Kubernetes, "Kubernetes: Production grade container orchestration," 2021. [Online]. Available: https://kubernetes.io/

[19] Istio, "Istio," 2021. [Online]. Available: https://istio.io/latest/

[20] E. V. Mangipudi, K. Rao, J. Clark, and A. Kate, "Towards automatically penalizing multimedia breaches (extended abstract)," in *2019 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2019, Stockholm, Sweden, June 17-19, 2019*, 2019.

[21] M. C. Vulnerabilities and Exposures, "Cve-2014-6407," 2014. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6407

[22] ——, "Cve-2014-9357," 2021. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9357

[23] ——, "Cve-2015-3631," 2015. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3631

[24] U. L. Package, "Lxc sysrawio abuse," 2015. [Online]. Available: https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1511197/

[25] M. C. Vulnerabilities and Exposures, "Cve-2015-3627," 2015. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3627

[26] "API endpoints are vulnerable to SQL injection," September 2020, https://jira.hyperledger.org/browse/BE-833.

[27] "Couchdb name collisions due to similar named channels," February 2017, https://jira.hyperledger.org/browse/FAB-2487.

[28] "ECDSA signature validation vulnerability by accepting wrong ASN.1 encoding in jsrsasign," June 2020, https://jira.hyperledger.org/browse/FABN-1585.

[29] "Cwe: Cwe-502 vulnerability in deserialization of untrusted data in google guava," November 2018, https://jira.hyperledger.org/browse/FABJ-390.

[30] "Fabric sdk node memory exposure vulnerability," July 2019, https://jira.hyperledger.org/browse/FABN-1290.

[31] G. Shaw, "Fabric security assessment management report," September 2017, https://wiki.hyperledger.org/download/attachments/13861997/management_report_linux_foundation_fabric_august_2017_v1.1.pdf.

[32] I. Blockchain, "Marbles chaincode," 2021. [Online]. Available: https://github.com/IBM-Blockchain-Archive/marbles

[33] Hyperledger, "Hyperledger caliper," 2021. [Online]. Available: https://www.hyperledger.org/use/caliper

[34] N. Narula, W. Vasquez, and M. Virza, "zkledger: Privacy-preserving auditing for distributed ledgers," in *Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[35] G. Zyskind, O. Nathan, and A. Pentland, "Enigma: Decentralized computation platform with guaranteed privacy," *arXiv*, vol. abs/1506.03471, 2015. [Online]. Available: http://arxiv.org/abs/1506.03471

[36] F. Benhamouda, S. Halevi, and T. Halevi, "Supporting private data on hyperledger fabric with secure multiparty computation," *IBM Journal of Research and Development*, vol. 63, no. 2/3, pp. 3:1–3:8, 2019.

[37] H. Zhong, Y. Sang, Y. Zhang, and Z. Xi, "Secure multi-party computation on blockchain: An overview," in *Parallel Architectures, Algorithms and Programming*, H. Shen and Y. Sang, Eds. Springer, 2020, pp. 452–460.

[38] R. K. Raman, R. Vaculín, M. Hind, S. L. Remy, E. K. Pissadaki, N. K. Bore, R. Daneshvar, B. Srivastava, and K. R. Varshney, "Trusted multi-party computation and verifiable simulations: A scalable blockchain approach," *arXiv*, 2018. [Online]. Available: http://arxiv.org/abs/1809.08438

[39] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 418–429.

[40] M. Zhang, D. Marino, and P. Efstathopoulos, "Harbormaster: Policy enforcement for containers," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 355–362.

[41] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 218–233.

[42] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: making linux security frameworks available to containers," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1423–1439.

[43] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, 2020, pp. 443–458.

[44] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, vol. 57, no. 72, 2004.

[45] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner, "Diesel: Applying privilege separation to database access," in *Proceedings of the 6th ACM symposium on information, computer and communications security*, 2011, pp. 416–422.

[46] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in html5 applications," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 429–444.

[47] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song, "Data-confined html5 applications," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 736–754.

[48] M. Krohn, "Building secure high-performance web services with OKWS," in *USENIX Annual Technical Conference*, 2004.

[49] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012, pp. 71–72.