# A Novel Architecture for Cyber-Resilient Self-Protecting Systems Based on Blockchain

Tommaso Caiazzi*, Stefano Iannucci*, Valerio Marini*, Diego Pennino†, Maurizio Pizzonia*, Riccardo Torlone*
*Roma Tre University, †University of Tuscia

***Abstract*—Self-Protecting Systems (SPS) rely on an autonomic manager to detect and mitigate cyber threats. However, a major challenge in SPS design is ensuring the security of the autonomic manager itself, as its compromise could lead to complete control of the system by an attacker. In this work, we propose a cyber-resilient SPS architecture that leverages permissioned blockchain technology to enhance the trustworthiness of both Intrusion Detection (ID) and Intrusion Response (IR). The proposed architecture is technology-agnostic and adaptable to various ID and IR techniques. We implement a prototype using Quorum and a smart contract and evaluate its performance in terms of overhead and scalability. Experimental results show that the proposed architecture is technically feasible and that it introduces a minimal overhead with respect to non-smart contract-based transactions, and that it can be used on production systems with high event rates despite the inherent scalability issues deriving from the usage of the chosen blockchain technology.**

***Index Terms*—Self-Protecting Systems, Blockchain, Intrusion Detection, Intrusion Response**

## I. Introduction

Self-Protecting Systems (SPS) are typically architected with two macro-components, namely: a managed system and an autonomic manager [1]. The former is the production system, while the latter is a controller that is in charge of detecting and possibly executing defense actions should the managed system found to be compromised. In particular, the autonomic manager of a SPS usually relies on a pipeline of two macro-components, namely: one or more Intrusion Detection Systems (IDS), in charge of detecting possible attacks directed to the managed system; and one or more Intrusion Response Systems (IRS), in charge of selecting the appropriate response to be executed to counter or mitigate it.

However, one of the most important critiques that is usually moved against SPSs is the possibility for the autonomic manager itself to be attacked, thus allowing the attacker complete control over the production system. This issue was already envisioned by Chess et al. [2] when, in a seminal work regarding cybersecurity for autonomic systems, on the one hand, they highlighted the benefits that the autonomic computing paradigm could have on system security; on the other hand, they discussed some emergent security issues introducing the need for trustworthy autonomic managers.

Designing a cyber-resilient autonomic manager is still an open challenge, as also highlighted in more recent works [3], [4]. For this reason, in this paper, we investigate the design of an architecture for cyber-resilient SPSs. In particular, we focus on the need for trustworthy detection of cyber-attacks and trustworthy selection of response actions. To this end, we employ a permissioned blockchain technology (i) to keep track of the events generated by the IDSs and achieve consensus on the actual state of the system (*i.e.*, whether or not the system is under attack); and (ii) to keep track of the response action proposals and achieve consensus on which one has to be executed to counter the detected attack. Using this strategy and a Byzantine Fault Tolerance (BFT [5]) consensus, the autonomic manager can withstand the compromise of up to $1/3$ of the IDSs and up to $1/3$ of the IRSs under the assumptions and according to the threat model detailed in Section II.

It is worth noting however that our proposed architecture is independent of the specific blockchain technology, and from the adopted IDS and IRS techniques. Indeed, for simplicity, we realized an openly accessible instance of SPS prototype [6] following our architecture using Quorum and the QBFT [7] consensus algorithm, where events are synthetically generated and responses are selected from a static state-action map. The performance evaluation conducted using the prototype shows that the overhead introduced by the intrusion response smart contract is negligible. Moreover, it demonstrates the applicability of the proposed approach even to systems with large action maps. Finally, while on the one hand it highlights potential scalability issues on the blockchain leader due to the serialization of the Intrusion Detection events, on the other hand it shows that the rate of sustainable events using a single-CPU Virtual Machine is high enough (nearly 500 events/sec) for systems with high event rate generation.

The paper is organized as follows: the architecture model and the threat model are described in Section II; the prototype implementation is illustrated in Section III; experimental results are discussed in Section IV; in Section V related works in the field of SPS and cyber-resilience are presented. Finally, conclusions are drawn in Section VI.

## II. Models

In this section, we first describe our proposed architecture and the role of the blockchain. Then, we describe our threat model.

### A. System Architecture Model

The proposed SPS architecture is represented in Figure 1. We assume both the manager system and the managed system to be made of several *components* that run in *containers* that in turn are hosted in physical or virtual *hosts*. The isolation provided by these containers is crucial to ensuring the security

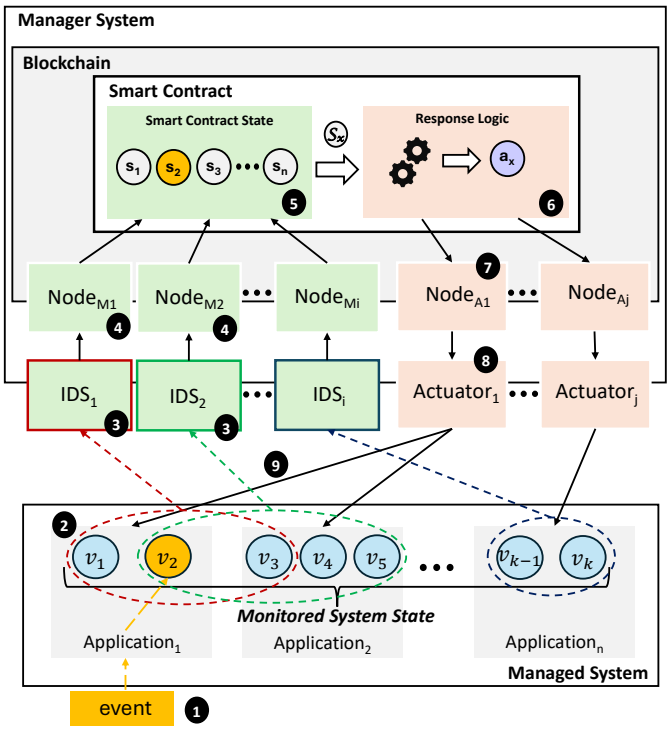


Fig. 1: The Proposed SPS Architecture

of our approach, as will become evident in Section II-B. Components of the managed system are *applications* to be controlled by the manager system, and hence, their containers are named *application containers*. The application itself and the containers it runs can be monitored with respect to several aspects. In general, for each container, there are several *state variables* (*e.g.*, CPU usage, disk I/O) that change over time and are monitored by the manager system. The set of all the monitored variables is named *monitored system state*. Referring to a collection of variables to identify the system state is typical in the field of intrusion response, e.g., [8].

The manager system is made by IDSs, *actuators*, and a *blockchain* made of several *blockchain nodes*. Each IDS, each actuator and each blockchain node runs in its own container. The blockchain and the actuators form the IRS, which is part of the manager system. Each IDS and each actuator is paired with its own blockchain node, which is used to interact with the blockchain. Additional blockchain nodes may be possibly present in the manager system to improve the overall resiliency. In fact, the resiliency of a blockchain typically depends on the number of component nodes.

The SPS architecture proposed in this work is independent of the specific blockchain technology. However, it should have the following characteristics. $(i)$ It should be private and permissioned so that we have full control over the identity of nodes and their ability to join the network. $(ii)$ It should handle transactions quickly. $(iii)$ It should provide the ability to implement the response logic (described in the following) in a smart contract.

A `smart contract` is a self-executing program deployed on a blockchain that automatically enforces predefined rules and agreements between nodes, without the need for intermediaries. Once deployed, smart contracts run deterministically across all nodes in the network, ensuring transparency, trustlessness, and immutability of the encoded logic.

The blockchain leverages on a smart contract to keep a so-called *smart contract state* that is shared among all the IDSs and all the actuators. This state contains a version of the monitored system state, created on the basis of information provided by the IDSs, plus additional information on the internal state of the IRS itself (e.g., whether a given response has already been executed). The blockchain is also in charge of applying a response logic to decide which response, if any, should be acted by actuators. Each IDS monitors a subset of the state variables and contributes to updating the shared view of the monitored system state by sending regular blockchain transactions. At a certain instant of time, IDSs may not agree on the state, for example, because one of them was compromised or because they have different detection times. A shared consistent view of the monitored system state is obtained by a voting approach. For this reason, in our architecture, the same variable is preferably monitored by more than one IDS (ideally, at least three). Further, IDSs should be *heterogeneous* regarding their technology (see Section II-B). A way to realize the voting approach to reach a shared state and to safely compute a response according to a response logic is to adopt a smart contract. A *smart contract* is a software object running in a blockchain, for which, the integrity of its state and of its execution is guaranteed by the integrity properties of the blockchain itself. Interactions with the smart contract are performed by the IDS by *transactions*. In a transaction, an IDS declares its view of one or more monitored state variables. The smart contract decides when to update the shared system state on the basis of the consensus on the current monitored state, given the state updates received from the IDSs. When the shared state changes, the smart contract applies a response logic to select a possible response. This can be thought of as a generic function of the state. In its simplest implementation, supposing a state with the Markov property, i.e., a state that contains all the needed information and the path that led to the state is not relevant for future transitions, it is just a table that maps states to actions. The result of the application of the response logic is recorded in the smart contract state, making clear to all the components of the system that the manager system is responding in some way. This is useful for three reasons: $(i)$ actuators read directly from the smart contract state if they have to actuate any response, $(ii)$ the effect of the response may be expected by IDSs, and $(iii)$ response logic itself may signal any unexpected behavior (*e.g.*, an unapplied response).

Changes in the smart contract state are known to all nodes since accepted transactions are packed into blocks that are broadcast to all blockchain nodes. Consequently, state changes applied by the response logic can be used to trigger the appropriate countermeasure. In this way, blockchain nodes

associated with actuators are informed about responses to actuate through regular blockchain broadcast and can relay them to actuators. For security reasons, we assume that the actuators are *blind*, in the sense that they perform their task without reading any state. Note that, this does not prevent them from obtaining state-related information from an IDS through the blockchain if needed. Figure 1 also shows the sequence of events triggered by a change in the monitored system state. An event ❶, either malicious or not, changes the system state ❷. One or more IDSs ❸ recognize them and send a transaction to their blockchain node ❹ containing the corresponding update of the smart contract state. The smart contract accumulates and changes the corresponding state variables in the smart contract state ❺ when the majority of IDSs asks for it. If the state change requires an action according to the response logic ❻, a random actuator, chosen from those capable of performing the required action, is assigned to execute it and the associated node is notified ❼. Actuators ❽ are notified through push mechanisms, which are commonly supported by blockchain technologies [9]. Actuators perform ❾ their response action.

### B. Threat Model

In this section, we describe our threat model.

The purpose of the SPS architecture described in Section II-A is twofold: $(i)$ to provide detection and response capabilities to enable automatic protection of the managed system in case of attacks on the latter; $(ii)$ to be resilient to attacks against the autonomic manager itself.

While the first objective is common to any SPS, we focus on the second one. From our point of view, a *successful attack* is an attack on the autonomic manager that compromises the correct behavior of the manager itself. More specifically, an incorrect behavior of the manager is one of the following.

1) An attack is detected, but the response dictated by the response logic is not executed.
2) An actuator performs certain actions that are not triggered by detection, potentially going beyond what is typically considered a response.

We now describe the capabilities of the attacker and the assumptions on the security of the elements of our architecture.

We assume that the attacker cannot attack any (physical or virtual) host directly. Further, an attacker cannot escape a container to get into the host. This means that the only way to attack a container is to attack any process already running in the container using one of its input channels. We do not make any security assumptions about application processes. By attacking them, an attacker can get into an application container.

Regarding the manager system, we assume all the communications among its containers to be well isolated so that they do not receive any input besides those encompassed by the presented architecture. In other words, the only components that directly receive untrusted input are the IDSs. Coherently with the above assumption, actuators were defined to be *blind* in Section II-A. This means that an actuator cannot be directly attacked from the application container intended to act upon. Given these assumptions, any attack on the autonomic manager must first attack an IDS from an application container.

Smart contracts may be vulnerable. There is a large literature about smart contracts vulnerabilities. For example, for the solidity technology adopted in the implementation described in Section III, they are surveyed in [10] and [11]. We assume that the smart contract in the manager system is correctly and securely realized.

To force the autonomic manager to behave incorrectly, the attacker can adopt one of the following strategies.

1) provide wrong inputs to the response selection logic,
2) tamper with the execution of the selection by taking control of the blockchain itself,
3) change the behavior of the actuators so that they do not execute the correct response, do not execute anything, or execute a completely unrelated action.

We will now briefly discuss the challenges an attacker faces when pursuing each of the three strategies within the context of our assumptions and architecture.

Regarding Strategy 1, the voting mechanism described in Section II-A acts as a countermeasure against the compromise of a single IDS. It forces the attacker to compromise the majority of IDS containers. This is hard if each state variable is monitored by at least three *heterogeneous* IDSs, where we suppose that if the attacker can find a vulnerability in one of them, the other two can still monitor and provide good inputs to the selection logic correctly. Note also that, some IDSs might perform monitoring only by getting inputs from outside the container. This typically does not provide the attacker with a viable channel to execute an attack, resulting in significantly higher security.

Regarding Strategy 2, the consensus mechanism underlying a blockchain acts as a countermeasure. This is usually a *Byzantine Fault Tolerant* (*BFT*, [5]) consensus that is resilient up to $n/3$ malicious nodes. However, to attack one blockchain node, the attacker should first attack its corresponding IDS, thus reaching this target may be very hard for the attacker. Subversion of the blockchain might be achieved by the attacker by spreading the attack through the whole blockchain network. Special care should be taken to make this hard to achieve (for example by adopting heterogeneous node implementations). Further, additional nodes not connected to any IDSs can be provided to make the blockchain more resilient, raising the number of nodes to be compromised for a successful attack.

Regarding Strategy 3, we point out that actuators cannot be directly attacked due to their blindness. Hence, the attacker should necessarily pass through the blockchain.

### C. Threat Model Discussion

In this section, we discuss strengths, limits and variations to our threat model.

First, we note that our setting is a permissioned one, i.e., nodes authenticate each other. For this reason, many of the attacks that are traditionally known to be critical for unpermissioned blockchains (like Eclipse and Sybil attacks) cannot be performed. Another possible source of concern is the

vulnerability of the blockchain implementation itself. Usually, this mostly regards the exploitation of a wrongly realized RPC API of the node that a malicious client can leverage to compromise it and possibly propagate the attack through neighbours. First, note that in our threat model the attacker cannot directly provide input to a node unless it has already successfully attacked an IDS. Second, the risk of propagation can be mitigated by having nodes with heterogeneous realization flavours for the node software, for example, having one flavour dedicated to only edge nodes and a different flavour dedicated to the rest of the nodes. In this way, the attacker can compromise only one edge node and cannot use the same exploitation technique to subvert the rest of the network.

Particular care should be taken to dimension autonomic manager nodes, and/or limit requests, so that available resources are enough to sustain the transaction workload.

Our architecture is not resilient to attacks that can act on containers and communications from the host. In fact, in that case, an attacker can perform several kinds of attacks. For example, it can shutdown IDSs to interrupt inputs to the autonomic, while shutting down actuators makes the autonomic manager ineffective. The same effects can be obtained by interrupting the corresponding communication channels. Further, consensus manipulation could be performed by shutting down nodes or tampering with at least 1/3 of the nodes.

On the contrary, our architecture can be slightly modified to tolerate a less stringent threat model. Suppose to deploy the autonomic manager nodes on more than one host and suppose that hosts are heterogeneous, which allows us to assume that the attacker can successfully compromise only one of them. In this case, if the fraction of nodes of the autonomic manager deployed on the same host is less than 1/3, it is easy to prove that our approach is still secure.

## III. Implementation

To demonstrate the feasibility of the proposed architecture, we implemented a prototype of the model presented in II-A. The objective is not to showcase a complete use case, but rather to assess feasibility. Therefore, in our prototype, we focus solely on modeling blockchain components while mocking the interactions between the blockchain, the IDSs and the managed system. We reserve the implementation of a complete scenario for future work.

We realized our prototype using the Quorum blockchain technology, which satisfies the requirements described in Section II-A. Among the consensus algorithms available in it, we selected QBFT [7], for its resilience and security features. Additionally, we considered RAFT [12] as an efficient benchmark, though it lacks tolerance to malicious attacks. Note that the overall performance and scalability of the system heavily depend on the underlying blockchain technology and the consensus algorithm selected for implementation. However, the objective of this work is not to deliver a production-grade solution, but rather to analyze the methodology and assess the validity of the proposed idea. Indeed, other blockchain technologies with efficient Byzantine fault-tolerant consensus mechanisms (*e.g.*, Algorand [13]) exist and should be explored. We leave their evaluation for future work.

In our experimentation [6], the smart contract is written in Solidity, a well-known de-facto standard for smart contract development. It keeps a key-value store for the representation of the monitored system state called *parameter-status* map. The parameter-status map stores as key an identifier of the state variable of the state, and as value the current value declared by the majority of monitoring nodes in the system. The smart contract also represents the response logic with a *state-action* map. It stores as key a representation of a system state, and as value the action required to protect the system found to be in that state. Note that, the state-action map might contain a huge number of different states, as they represent permutations of the status parameters. Such a map cannot be written directly into the contract at the time of the deployment, as this would exceed the maximum size allowed for a single transaction. This limit may differ slightly between different blockchain technologies. To solve this problem, a map population function was implemented, which can only be invoked by the owner of the contract (i.e., the node that deployed it).

The smart contract allows the IDSs to update its state by means of a function named *proposeNewValues*. The IDS that detects the change of a state variable invokes the function *proposeNewValues* by issuing a suitable blockchain transaction, passing the list of modified state variables along with their new values. Each invocation of this function keeps track of the proposed values and, for each of them, checks if that value is proposed by a majority of the IDSs observing that state variable. If this is the case, the function updates the representation of the monitored system state and checks whether the new state requires an action, using the state-action map. If an action is required, an actuator node is arbitrarily selected. We use the event/emit construct of Solidity to push the event to the selected actuator.

All the blockchain nodes are modeled as containers using the Kathará [14] network emulator.

## IV. Evaluation

The evaluation of the effectiveness of a SPS involves assessing its two main macro-components: Intrusion Detection and Intrusion Response. While the former benefits from an established evaluation methodology—typically based on ground truth data from purpose-built datasets—the latter remains more challenging to assess. Indeed, an Intrusion Response System cannot be reliably evaluated using datasets alone, as each response action taken to protect the system can alter its subsequent behavior, rendering any predefined attack trace obsolete. This challenge is addressed in [15], which proposes a methodological framework for the empirical evaluation of SPSs. Specifically, the work highlights the importance of a formal definition of security policy and introduces a set of metrics to assess SPS quality. These include the duration of an attack, the time required for detection and response, and the cost associated with the response. The overall effectiveness of

an SPS is then measured in terms of the performance gap it can achieve relative to manual protection.

In this work, our focus is not on securing a specific system, but rather on proposing an SPS architecture that is inherently resilient to cyber-attacks. Consequently, our evaluation considers performance metrics that directly influence attack duration, detection time, and response time. In particular, we evaluate the prototype presented in Section III, addressing the following questions.

Q1. *How many transactions can the system process?*
Q2. *How does handling the smart contract affect latency?*
Q3. *How does the transaction size impact latency?*
Q4. *How does the state-action map size impact throughput?*

**Testing Environment.** In order to run the experiments, we developed a system based on the Kathará network emulator to generate and configure network scenarios using parameters to control different dimensions (*e.g.*, the number of validators, the number of actuators, the transactions per second), while tracing the results. In all the experiments, the blockchain is composed of three Validators (nodes that participate in the consensus protocol), and eight Members (nodes that do not participate in the consensus protocol), five acting as monitoring agents and three as actuators. In the QBFT consensus, the leader node, i.e., the one that proposes the next block, is chosen using a Round-Robin strategy. In the RAFT consensus, the leader remains unchanged throughout the experiment. All the nodes are directly connected. The system monitors 25 state parameters and the time between consecutive blocks (*blocktime*) is set to 1 second. The choice of these parameters has been made to represent a simplified scenario, suitable for studying the feasibility. It is not intended to suggest that these values are optimal or appropriate for real-world scenarios, for which detailed analysis is required and left to future work. To analyze how the smart contract and the state-action map impact performance, we run the experiments using four different systems configurations: $(i)$ a system, based on RAFT consensus, that only performs cryptocurrency transfer transactions that do not involve running the IRS smart contract (`baseline-raft` in the following); $(ii)$ an IRS, as described in Section III, with an action map of 1 K entries based on RAFT consensus (`contract-raft` in the following); $(iii)$ a system, based on QBFT consensus, that only performs cryptocurrency transfer transactions that do not involve running the IRS smart contract (`baseline-qbft` in the following); $(iv)$ an IRS, as described in Section III, with an action map of 1 K entries based on QBFT consensus (`contract-qbft` in the following). All the experiments are repeated 5 times, tracing each system's performance for 40 seconds. Average and confidence intervals are shown in the figures. To ensure fair and consistent results across different runs and consensus algorithms, we limit each validator to a single CPU. Future work will explore the scalability of the architecture in terms of available resources per node and the number of nodes participating in the blockchain.

**Testbed.** The experiments are performed on a single virtual machine configured with 24 cores and 64 GB of RAM running Ubuntu 24.10 as the guest operating system. The underlying physical machine was equipped with an Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz and 1 TB of RAM.

**Reproducibility.** We publicly release the code needed to reproduce all the experiments. All the materials related to the paper are available at [6].

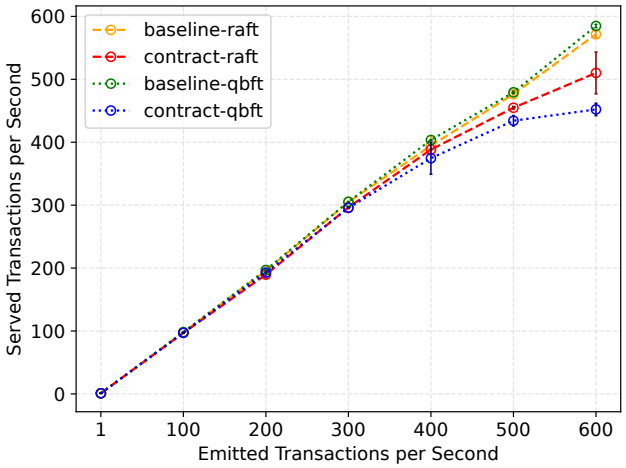


Fig. 2: Served transactions rate over emitted transactions rate.

**Q1: Up to 500 transactions per second on a single CPU.** To determine the maximum transaction throughput supported by the system, we measure the rate of successfully processed transactions while increasing the rate of incoming transactions. Figure 2 illustrates the number of served transactions per second (y-axis) as a function of the emitted transactions per second (x-axis). Each transaction changes a single value of the state parameters. Both baseline systems, which handle standard cryptocurrency transfer transactions, can sustain the input throughput across all scenarios. In contrast, the IRS systems must process smart contracts, increasing the computational overhead and impacting throughput. Specifically, the `contract-qbft` system can sustain up to 450 tx/s, and, for higher rates, it starts performing worse than baseline systems. Notably, this is only slightly lower than the `contract-raft` system ($\sim$ 500 tx/s), which, however, lacks resilience to Byzantine faults. Nevertheless, considering that this throughput represents the number of alerts from IDSs that the system can handle, a rate of 450 tx/s is sufficient for many use cases.

**Q2: Minimal latency in contract handling under low load conditions.** To understand the latency overhead introduced by the contract and how the input transaction throughput affects the latency, we measure the latency of served transactions, from submission to acceptance in a block, while increasing the rate of the emitted ones. Figure 3 shows the average latency of served transactions in seconds (y-axis) while varying the emitted transactions per second (x-axis). Each transaction changes the value of a single state parameter. In all the scenarios, the average transaction latencies tend to be a multiple of the blocktime that in our experiment is set to one second. We

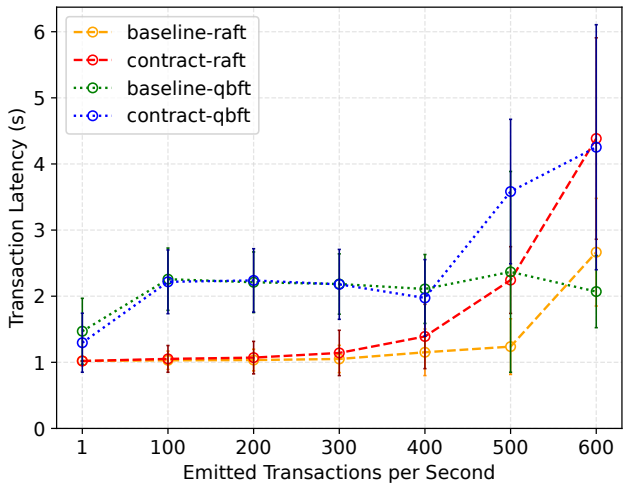


Fig. 3: Served transactions latency over emitted transactions rate.

note that the adoption of QBFT has the effect to step up the latency of about one blocktime. However, the latency remains roughly constant until the emitted transaction rate is below the maximum throughput (which for us is about 400 tx/s). Beyond this point, the blockchain transaction pool starts to grow, causing some transactions to remain in the pool for multiple block rounds, thereby increasing the average latency. However, we note that, until the system is able to maintain the input throughput, a latency of two seconds may be considered acceptable in most situations.

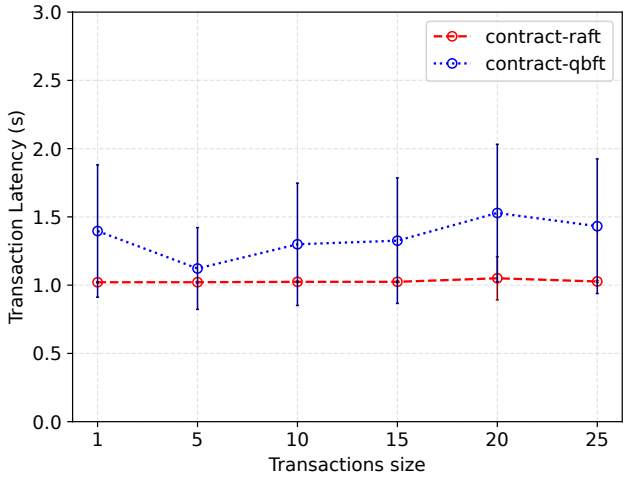


Fig. 4: Served transactions latency while increasing transactions size.

**Q3: Increasing transaction size has a minimal impact on latency.** To evaluate the impact of the transaction size (*i.e.*, the number of state parameters modified by a transaction) on latency, we measure transaction finalization times while gradually increasing their size. In this experiment, we omit the baseline, as transaction size is not a meaningful concept in that context. All the experiments are conducted at a fixed rate of one transaction per second. Figure 4 illustrates the average latency of processed transactions (y-axis) as the transaction size varies (x-axis). The results indicate that the latency remains largely stable as the transaction size increases for both consensus mechanisms. As expected from previous findings, the `contract-qbft` system incurs slightly higher overhead compared to `contract-raft`. This stability enables monitoring nodes to send IDS alerts in batches, effectively increasing the number of alerts the system can handle per second. As future work, we plan to analyze the impact of transaction size at higher rates to determine the optimal trade-off for specific use cases.

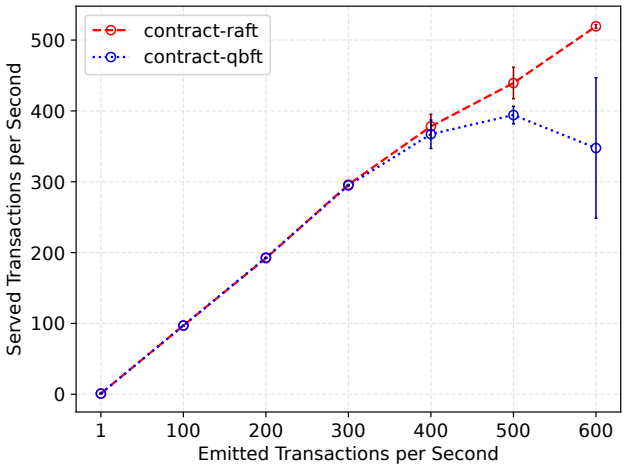


Fig. 5: 100 K state-action map: served transactions rate over emitted transactions rate.

**Q4: Slightly decrease under high load conditions as the state-action map size increases.** As an additional test, we repeated the experiments from Q1 using different state-action map sizes. Specifically, we increased the map size up to 100 K entries, verifying that the overall results remained consistent with previous experiments. Figure 5 presents the results for a map size of 100 K. While the `contract-raft` line shows no significant degradation, the `contract-qbft` line exhibits a slight performance decline as the transaction rate increases. However, under normal load conditions, the map size does not impact the rate, demonstrating that the system effectively supports larger state sizes.

## V. Related Work

In the field of Self-Managing Systems and, in particular, of SPS, only few works addressed the problem of protecting the autonomic manager. A broad classification of attack vectors to self-adapting systems is provided in [16], indicating that the majority of the attacks are due to the manipulation of system resources, deceptive interactions, injection of unexpected items, subversion of access control, collection and analysis of the information, and abuse of existing functionality. However, the need for targeted security mechanisms was raised nearly 10

years before, as reported in [3], where the authors emphasize the importance of investigating on *how to protect the protector*, that is, how to safeguard the meta-level self-protection module, especially in a globally decentralized topology.

Trustworthiness is widely recognized as a fundamental property to build secure, robust and dependable autonomous systems. A taxonomy of trustworthy autonomous systems is presented in [4], where the authors introduce the concept of *cyber-resilience* as an extension to cyber-security, focusing on the strategies and on the policies that should be adopted to support a self-adapting system despite unexpected operating conditions, which may include cyber-attacks. One possible implementation of a cyber-resilient system is described in [17], where the design of a trustworthy self-adaptive software is based on assurance cases, allowing the formal verification of properties, however without providing a specific, secure by design perspective on how to realize an autonomic manager. For this reason, this work can be considered complementary to ours, and in particular, the formal verification of system properties through assurance cases could be seen as an extension to be applied on top of the approach proposed in this work.

In [18], the architectural design of a self-protecting IoT system leveraging blockchain to guarantee the trustworthiness of the adaptation is presented. The blockchain is used to store details about the participating agents, e.g., public keys, MAC addresses, along with their beliefs regarding the system state and the corresponding action that must be executed. This work introduces a system design that is conceptually similar to what we propose in this paper. However, a prototype is not available and a threat model is not discussed, which makes the limitations of the proposed approach unclear.

Finally, there exist technologies that are somewhat related to the aims of our work. The *Trusted Platform Module* (TPM) is a hardware-based technology that is used to provide some form of platform integrity. The TPM is normally used to check that the boot process involves only software from trusted sources. This is not useful in our case, in which the attacker might compromise the system while it is running. A *Trusted Execution Environment* (TEE), sometimes also called *secure enclave*, is a hardware-based execution environment that is isolated from the rest of the system and protects sensitive computations from attacks to the system. This is essentially the same function played by the container isolation requirement in our threat model. The software that runs in the TEE may be attacked by usual means, in the sense that a vulnerability of the software that runs in the TEE can be exploited as usual. However, certain TEEs (e.g., Intel SGX) can perform *remote attestation*, providing a remote client with proof that the code running in the TEE is untampered. Usually, a TEE is used to run small specific software, and it may be unfeasible to run a whole autonomic manager within a TEE. A third technological option to consider is the adoption of an architecture similar the one proposed in Section II-A but with a consensus protocol used without the machinery related to the blockchain. Actually, the history-recording feature of the blockchain is not very relevant in our approach. On the contrary, the fact that the transactions are processed in blocks is relevant to achieving a high throughput. Hence, a solution adopting pure consensus will end up to be quite similar to a blockchain without history records.

## VI. Conclusion and Future Work

In this work, a novel architecture for cyber-resilient SPSs was proposed. We discussed the architectural model and the threat model, clarifying the potential limitations of the proposed approach. Experimental results show that while there are inherent scalability issues with the blockchain, the rate of events that can be sustained by the autonomic manager is high enough for most production systems.

As part of future work, we intend to extend the experimental evaluation to other consensus algorithms and blockchain technologies. Moreover, we plan to study the scalability issues related to the serialization of the alerts and devise a blockchain that can process transactions in parallel if there is an empty intersection of the variables involved in the state change. Finally, we plan on evaluating the performance of the autonomic manager while scaling the number of IDSs and blockchain nodes, possibly considering known approaches for blockchain scalability, e.g., the one described in [19].

## Acknowledgment

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU; by the PRIN 2022 project 2022Y45XE3 *Panacea: A Model-Based Framework for Self-Protecting Systems* funded by Ministero dell'Università e della Ricerca (MUR); and by Progetto Integrato 2.1 "Cyber Security dei sistemi energetici" PTR22-24 funded by Ministero dell'Ambiente e della Sicurezza Energetica (MASE).

## References

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[2] D. M. Chess, C. C. Palmer, and S. R. White, "Security in an autonomic computing environment," *IBM Systems journal*, vol. 42, no. 1, pp. 107–118, 2003.

[3] E. Yuan, N. Esfahani, and S. Malek, "A systematic survey of self-protecting software systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 8, no. 4, pp. 1–41, 2014.

[4] F. Flammini, C. Alcaraz, E. Bellini, S. Marrone, J. Lopez, and A. Bondavalli, "Towards trustworthy autonomous systems: Taxonomies and future perspectives," *IEEE Transactions on Emerging Topics in Computing*, 2022.

[5] X. Wang, S. Duan, J. Clavin, and H. Zhang, "Bft in blockchains: From protocols to use cases," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–37, 2022.

[6] T. Caiazzi, D. Pennino, V. Marini, S. Iannucci, M. Pizzonia, and R. Torlone, "irs-blockchain-gitlab," 2025. [Online]. Available: https://gitlab.com/uniroma3/compunet/networks/irs-blockchain

[7] "Qbft — consensys goquorum," 11 2023, [Online; accessed 2025-02-26]. [Online]. Available: https://docs.goquorum.consensys.io/configure-and-manage/configure/consensus-protocols/qbft

[8] S. Iannucci, S. Abdelwahed, A. Montemaggio, M. Hannis, L. Leonard, J. S. King, and J. A. Hamilton, "A model-integrated approach to designing self-protecting systems," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1380–1392, 2018.

[9] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, vol. 1, pp. 22–23, 2013.

[10] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust*. Springer, 2017, pp. 164–186.
[11] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Systematic review of security vulnerabilities in ethereum blockchain smart contract," *Ieee Access*, vol. 10, pp. 6605–6621, 2022.
[12] Quorum, "Raft consensus overview - quorum." [Online]. Available: https://goquorum.readthedocs.io/Consensus/raft/raft/
[13] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.
[14] M. Scazzariello, L. Ariemma, and T. Caiazzi, "Kathará: A Lightweight Network Emulation System," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020.
[15] A. Montemaggio, S. Iannucci, T. Bhowmik, and J. Hamilton, "Designing a methodological framework for the empirical evaluation of self-protecting systems," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2020, pp. 218–223.
[16] I. Pekaric, R. Groner, T. Witte, J. G. Adigun, A. Raschke, M. Felderer, and M. Tichy, "A systematic review on security and safety of self-adaptive systems," *Journal of Systems and Software*, vol. 203, p. 111716, 2023.
[17] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly, "Engineering trustworthy self-adaptive software with dynamic assurance cases," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1039–1069, 2017.
[18] F. Alkhabbas, M. Alsadi, S. Alawadi, F. M. Awaysheh, V. R. Kebande, and M. T. Moghaddam, "Assert: A blockchain-based architectural approach for engineering secure self-adaptive iot systems," *Sensors*, vol. 22, no. 18, p. 6842, 2022.
[19] G. D. Monte, D. Pennino, and M. Pizzonia, "Scaling blockchains without giving up decentralization and security: A solution to the blockchain scalability trilemma," in *Proceedings of the 3rd Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, 2020, pp. 71–76.