Multi-Stakeholder Policy Enforcement for Distributed Systems

Robert Walther
robert.walther@tu-dresden.de
TU Dresden
Germany

Carsten Weinhold
carsten.weinhold@barkhauseninstitut.org
Barkhausen Institut
Germany

Peter Amthor
peter.amthor@tu-ilmenau.de
TU Ilmenau
Germany

Michael Roitzsch
michael.roitzsch@barkhauseninstitut.org
Barkhausen Institut
Germany

Abstract

Cloud environments, comprising both virtual and physical servers, are complex distributed systems that require clear and expressive configuration descriptions. Human-readable configuration formats like Kubernetes YAML are state of the art, but they lack the granularity needed for fine-grained control and advanced policy enforcement. To address these limitations, we propose an abstract system description approach that incorporates additional application properties, enabling more sophisticated policy decision-making rather than relying on resource constraints and port-based network restrictions. Our framework introduces two modes of policy enforcement: one allows system designers to automatically verify and manipulate system descriptions before translating them into concrete configurations, while the other enables communication partners to review the descriptions for assessing trustworthiness. We introduce a user-friendly description language paired with an extensible policy enforcement engine, providing stakeholders with the ability to define deployment scenarios intuitively and securely. We demonstrate the suitability of the approach for three different platforms, ranging from an embedded system to state-of-the-art container runtimes, namely Kubernetes and Docker Compose.

CCS Concepts

• **Software and its engineering → Orchestration languages**;
• **Computer systems organization → Cloud computing**.

Keywords

scenario language, policy enforcement, application deployment

ACM Reference Format:
Robert Walther, Carsten Weinhold, Peter Amthor, and Michael Roitzsch. 2024. Multi-Stakeholder Policy Enforcement for Distributed Systems. In *10th International Workshop on Container Technologies and Container Clouds (WoC '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3702637.3702958

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
WoC '24, December 2–6, 2024, Hong Kong, Hong Kong
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1339-2/24/12
https://doi.org/10.1145/3702637.3702958

1 Introduction

In cloud environments, applications never run in complete isolation. Instead, they depend on the operating system (OS) for access to resources such as network and storage, as well as higher-level services offered by the cloud platform. Beyond this interaction with system-level components, an application may also cooperate with other programs. It is often the case that only applications deployed by the same user will communicate with each other. But there are also use cases where applications owned by multiple stakeholders must cooperate, even if they do not trust each other completely.

Deployment and Security Policies The requirements and constraints governing these interactions are often complex, necessitating solutions that can adapt to the evolving needs of all involved parties. For instance, deployment policies may dictate that applications owned by different users, say Alice and Bob, must not interact or share the same processor core. As regulations or operational needs change, these policies must evolve, adding another layer of complexity. Manually configuring systems to enforce such dynamic, multi-layered rules is not only cumbersome but prone to human error.

The Need for Automation There is a plethora of solutions that provide run-time access control mechanisms in cloud environments. However, to the best of our knowledge, none of them enables automatic policy enforcement during the configuration and deployment of distributed systems. In increasingly complex cloud environments, this gap leaves these systems vulnerable to configuration errors, which can lead to insecure or otherwise incorrect deployments. Furthermore, it is crucial to assess the trustworthiness of applications and the services they depend on, before these components start to communicate with each other or external entities. So far, this assessment has largely been done through certificate checks or remote attestation. However, on their own, these approaches lack the ability to express nuanced security policies, because they reduce all configuration details to an opaque hashsum.

Package managers for operating systems already solve a similar problem for programs running on a single machine. They track dependencies and version requirements, and some also support sandboxing via containers. However, no comprehensive tools exist to manage the deployment and policy enforcement for groups of applications and services across distributed systems. A solution that automates these processes is needed to ensure security and functional compliance without risk of human error.

Gaps in Expressiveness Kubernetes and Docker allow for basic network or resource allocation policies out of the box, but they do not support evaluating arbitrary policies based on application metadata. To address this limitation without altering these systems, an abstraction layer can be added on top of the low-level configurations. This layer enhances policy granularity by incorporating richer metadata about applications, services, and communication channels. Various policy languages have been developed for a wide range of use cases, but many focus on role-based access control (RBAC). As a result, they are limited in describing communication channels or reasoning about attributes such as version numbers. Adapting these languages to support deployment and communication in distributed systems would require major modifications. In doing so, there is a risk that language concepts and restrictions that are critical for reasoning and formal guarantees are violated.

Contribution In this paper, we propose a comprehensive policy enforcement mechanism for distributed systems, utilizing an abstract system description approach. We introduce a simple and intuitive scenario and policy description language that efficiently expresses the communication requirements and constraints of multi-program application deployments. This language is supported by a platform-independent policy enforcement engine, which verifies if the described scenario meets all specified requirements. The engine operates in two distinct modes: deployment and validation.

In deployment mode, the engine checks a scenario description against policy constraints, and if all conditions are met, it automatically translates the description into a platform-specific configuration for deployment. In validation mode, a third party can use the engine to review a system's scenario description to assess its compliance with a security policy before initiating communication. The concept enables stakeholders, including regulatory bodies and infrastructure providers, to impose fine-grained rules on distributed systems without concerning themselves with the underlying cloud infrastructure.

Paper Outline In section 2, we first discuss background and terminology. We then describe the design of the language and the enforcement engine (section 3–4), before we show in a case study how our generic solution can be adapted to multiple target systems (section 5). In section 6, we compare our solution to related work. We conclude this paper with an outlook on future work (section 7) and a summary (section 8).

2 Background and Terminology

In this section, we provide more information on the context and scope of our work.

Policy Rules We address the problem of automatically configuring a computer system such that it runs one or more applications and their required system services according to a *policy*. We define such a policy as a set of *rules* that can express both functional requirements and auxiliary constraints. A requirement could be that application A must be able communicate with a service S. A constraint could be that this application A must not share a processor core with another application B.

Multi-Stakeholders Policies Requirements and constraints can be defined by various entities. The dependency of application A on service S will typically originate from the application vendor, who knows that A cannot function without S. But a system administrator may need to override the minimum required version for S, for example, because there is a security vulnerability in previous versions. Likewise, the constraint that applications A and B must not share cores may come from the system administrator, or it could also be demanded by a user who is wary of side-channel attacks. Cloud providers and regulatory bodies could impose rules, too.

We do not answer the question of who takes priority, if rules set by different stakeholders are in conflict. Such conflicts must be resolved outside the technical domain, but detecting unsatisfiable policies remains in scope of our work. We assume that requirements and constraints of all stakeholders are merged into a single set of rules. To keep the discussion in this paper focused, we therefore subsume the roles of users, administrators, and the other stakeholders in an abstract entity called *system designer*. We will refer to this entity in the following sections, but differentiate individual roles if needed.

3 Design

Our objective is to develop a policy enforcement and configuration generation engine. We explicitly target multi-application scenarios with all system-level and third-party services that those applications depend on. The system designer (subsuming multiple roles, as detailed in section 2) provides the application scenario and the associated policy, expressed using a description language.

Design Goals We set the following design goals for the overall solution:

- **Simplicity:** The description of applications and services to run must be simple. The same applies to the policy rules. Both must be easy to write and read in order to minimize the risk of configuration errors.
- **Expressiveness:** While maintaining simplicity, the description language should be expressive enough to articulate arbitrary relationships between applications and services, both positive and negative.
- **Composability:** Multiple stakeholders must be able to provide input for the scenario description, as well as additional policy rules. The resulting overall scenario and policy should be composed of these individual parts.
- **Separation:** Both the description language and the enforcement engine should be platform independent and not restricted to a single OS implementation or cloud environment. Furthermore, policy rules should be separate from the scenario description itself. This ensures reusability across a variety of systems and makes sure that policy rules can be reused for similar application scenarios.
- **Extensibility:** The system designer should be able to extend the policy and configuration language without modifying the implementation of the engine. This approach ensures that our solution can accommodate new requirements in the future.

With these five design goals, we target the following two main use cases. For both, the engine shall take as input: (1) a description of the applications and services to run, and (2) a policy describing the communication requirements and other constraints. This workflow is illustrated in Figure 1.

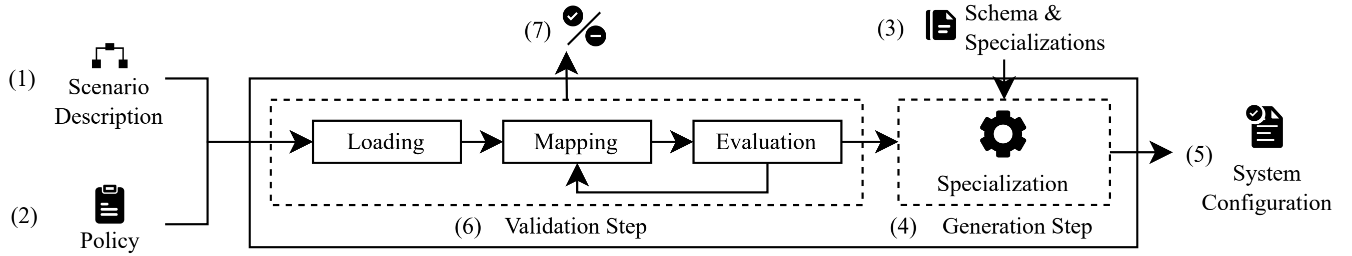

Figure 1: High-level workflow of the policy enforcement an configuration engine.

Deployment Mode For deployment, the system designer wants to automatically check specific rules against a new scenario description they intend to host, to prevent errors. In the deployment mode, the engine additionally consumes a platform-specific specialization template (3) that specifies the configuration format of the target system. Using these three inputs, the engine enforces the policies by either generating a conformant output configuration (4) or stopping the process. If successful, an administrator can apply the generated configuration directly to the target system (5).

Validation Mode In distributed computing scenarios, a communication partner may need to check if a multi-application scenario deployed on another machine conforms to a given policy. In such a case, this relying party receives the abstract scenario description from the remote system. The enforcement engine runs locally on the relying party's machine, where it will then validate in step (6) the received scenario description against a security policy chosen by this party. Steps (4) and (5) can be skipped in validation mode. The engine returns true or false, indicating whether the scenario description complies with the relying party's policy. In case of compliance, communication with the remote multi-program application can continue, or be terminated otherwise.

Trustworthiness of Deployments Both execution modes complement each other and can be combined with Trusted Execution Environments (TEEs) and remote attestation. In deployment mode, the relying party is the system designer, with validation happening automatically as part of the deployment process. On TEE-capable platforms, the scenario description and the policy may be protected against manipulation by enforcing a secure startup process. For validation performed by a remote relying party, a secure startup process that includes the enforcement engine and the scenario description may be required, too. It can provide evidence to the relying party that the remotely deployed scenario does indeed reflect what is being validated against the security policy. This kind of secure startup is orthogonal to the problem of describing and validating an application scenario against a policy. It is therefore not further discussed in this paper (but considered future work (section 7). We argue that our solution not only enhances manageability through fine-grained policy enforcement in distributed systems but also improves trustworthiness by enabling more detailed assessments of deployed configurations.

4 Policy Language and Enforcement Engine

We build our solution on the Lua programming language [1] because it offers an easily embeddable, imperative language which is essential for our configuration generation process. Lua's imperative nature, combined with its simple yet expressive syntax also allows nuanced policy rules to be expressed effectively. This enables us to use a single language for both generation descriptions as well as policy rules. In the following, we first introduce an example deployment. We will use the applications and services from that example to explain our scenario and policy description.

4.1 Example Scenario

Our scenario contains two applications *A* and *B* deployed in a cloud environment. They interact with a database *DB*. The database service stores data locally on the system. It is provided by company *C*. As the operator of *DB*, company *C* restricts access to the database to applications *A* and *B* that have also been manufactured by *C*. Due to a known vulnerability in versions of *A* prior to 4.0.0, application *A* should only be started, if the major version is at least 4.

4.2 Scenario Description

The description of an application scenario consists of two parts: (1) the applications and services that shall be started, and (2) a definition of communication channels between these programs. A configuration excerpt for our example scenario is shown in Listing 1.

Applications and Services Applications and services that shall be run must be listed in the `apps` table in the scenario description (line 1). For each program, the system designer must specify an identifier for the program (lines 2 and 8) and the program binary including any command-line arguments (via the `args` property in line 4). The system designer may add user-defined properties such as `manufacturer` (line 3) or `version` (line 5). These additional properties are not part of the language, but they can be referenced from policy rules. This satisfies the design goals for *Separation* and *Extensibility* and is further detailed in subsection 4.3.

Communication Channels To express communication requirements between applications or services, our description language provides the concept of channels. Lines 12–19 in Listing 1 describe such a communication channel. The mandatory `peers` property specifies that this channel shall be established between application *A* and the database *DB* from our example. Those programs are named using the identifiers declared under `apps`. The `type` property (line 13) declares what kind of communication channel to use.

Specialization Channel types are abstract and not predefined in the language. System designers can define them in a schema

```
apps = {
  db = {
    manufacturer = "company C",
    args = "db -l -p 6789",
    version = Ver("4.1.3")
  },

  app_a = {...}, ...
}

channels = {
  app_a_db_channel = {
    type = "channel_ipc",
    buffer_size = "1024MB",
    peers = {
      app_a = { is_initiator = true },
      db = { is_initiator = false }
    }
  }, ...
}
```

Listing 1: Example of scenario description

```
app_rules.app_a_version_check = {
  targets = { app_a },

  evaluator = function()
    return app_a.version >= Ver("4.0.0")
  end
}

channel_rules.db_version_check = {
  peers = { db, any },

  condition = function()
    return db.manufacturer == "company C"
  end,

  evaluator = function()
    return db.manufacturer
           == any.manufacturer
  end
}
```

Listing 2: Example of application and channel rules

file, allowing types like `datagram_ipc` (e.g., bi-directional message queue) or `multiparty_ipc` (e.g., shared memory between multiple programs). In practice, the platform manufacturer provides this schema file along with a specialization template to translate the abstract configuration into a concrete format for the target platform. Administrators and users only select the abstract type, without needing to understand the underlying mechanisms. This abstraction and specialization approach satisfies our design the goals of *Composability*, *Separation*, and *Extensibility*.

Similar to applications, the system designer can add arbitrary properties, such as a buffer size (line 14), to channel declarations. These properties can be either channel specific or peer specific and can be referenced from within the specialization template. For example, the `is_initiator` property (lines 16 and 17) indicates that application *A* should be capable of initiating the connection and sending requests, while the database service should only receive requests and respond.

4.3 Policy Description and Evaluation

In our description language, a policy consists of a set of rules. Each stakeholder can contribute such rules, thereby satisfying the *Composability* design goal.

Rule Types We differentiate these rules into two types: (1) *application rules* and (2) *channel rules*. The first type is applied only to the applications and services in the scenario description. The system designer can use them to control if and how applications are started. The second type of rule focuses on communication channels. The system designer can use channel rules to restrict which applications and services shall be allowed to interact with each other. A subset of the rules for our example scenario are shown in Listing 2.

Peer-based Rule Filters The policy rule restricting applications communication with the *DB* applies only to channels between the *DB* and other applications. To define filters for channel rules, we introduce the `peers` property (line 10 in Listing 2), which accepts a list of program identifiers or the `any` wildcard to match any program. For application rules, the `targets` property works similarly to `peers` but matches identifiers individually (match any), not as a group. Line 2 in Listing 2 specifies that the version check rule for application *A* from our example scenario shall apply only to application *A* rather than other programs.

Condition-based Rule Filters In addition to static rule filters based on the `peers` or `targets` property, our policy language also allows the system designer to formulate arbitrary filtering criteria in an optional `condition` function. An example is shown in lines 12–14 in Listing 2. For each application or channel rule that includes a `condition` function, the enforcement engine executes the Lua code in this function. If the function returns `true` and `targets` or `peers` match as well, the engine will apply the rule. Otherwise, the rule will be ignored.

Rule Evaluation All rules that remain after filtering are considered by the enforcement engine as the policy to be enforced. To determine if a rule can be satisfied, the engine runs an `evaluator` function, which the system designer provides for each rule. The `evaluator` is written in Lua and can perform arbitrary computations on the scenario application's and channel's properties. It returns `true` if the rule is satisfied, or `false` otherwise. For example, lines 17 and 18 in Listing 1 compare the `version` property of application *A*, accessed via Lua dot notation as `app_a.version`. We argue that Lua's simple syntax supports our *Simplicity* design goal while offering the *Expressiveness* needed for complex conditions.

Mapping Our description language treats the `any` keyword like a variable that can be instantiated with any program in the scenario description. Therefore, the policy enforcement engine may have to evaluate channel rules that use `any` multiple times in order to check all possible combinations of `peers`. The same is true for application rules that use `any` in their `targets` property. We call the process of instantiating a rule with concrete peers or targets the *Mapping* step. The engine performs this step before *Evaluation*, where it executes all `evaluator` functions.

Policy Enforcement and Rule Actions Depending on the mode, the engine performs the policy enforcement differently. In deployment mode, it may perform the mapping and evaluation multiple times, as shown in Figure 1. The engine exits once it found a scenario that satisfies all requirements expressed in the rules.

If the engine determines that the combination of all applicable rules cannot be satisfied, it aborts with an error and no target configuration is generated. In validation mode, the engine only performs the checking without manipulating or generating a final configuration, and yields only a `true` or `false` outcome.

4.4 Balancing Expressiveness and Security

Our policy enforcement engine is written in C++ but uses the Lua interpreter to parse and evaluate input files. Lua [1], an imperative language, is utilized in a declarative manner for the descriptive parts and imperatively for the logic parts. To limit its functionality to scenario and policy descriptions, we stripped down the Lua environment, ensuring the provided Lua code runs in a sandbox without OS [2] interaction or I/O [3] capabilities. We also disabled external module loading. A second attack vector is rooted in the Turing-completeness of Lua. A stakeholder could create erroneously or out of malice a rule that contains an endless loop in the `condition` or `evaluator` functions. In our prototype implementation, we mitigate this risk by enforcing an instruction limit [4] whenever the enforcement engine calls into such a function.

5 Case Study

We implemented a prototype of our policy enforcement engine and tested it with M³ [5], Kubernetes [6], and Docker Compose [7]. In accordance with our design goal for *Separation*, the implementation of the policy enforcement engine itself is platform independent. To add support for our test platforms, we assumed the role of the system manufacturer for each of the three target platforms and created Lua-based specialization templates for them.

M³ M³ is a microkernel-based OS with isolated user-level services and applications. To communicate with OS services like the network stack, explicit channels must be configured. The M³ platform relies on dedicated hardware support [8] to enforce both isolation and communication control. As multi-program scenarios are native to M³ and because of its strong isolation, deny-by-default approach, we argue that it is an ideal testbed for our policy enforcement solution, even though it primarily targets embedded platforms. Furthermore the platform is fully open-source[1], including a hardware simulator for testing. Using the scenario from subsection 4.2 and policy 4.3, we generated an M³ `boot.xml` file specifying program binaries and communication channels. Each application is placed on a dedicated core and the provided command-line arguments are passed to it. Communication channels are translated to M³'s *send* and *receive* gates.

Kubernetes Kubernetes [6] manages containerized applications by organizing them into services composed of interacting components named *pods*. It handles communication between pods through software-defined networking, enabling applications and services to interact across multiple machines. Since Kubernetes is a leading platform for orchestrating scalable and reliable distributed systems, we test our policy enforcement engine within this environment. Using our Kubernetes specialization template, the engine generates a loadable YAML configuration by translating our abstract program and channel descriptions into Kubernetes services. Communication channels are expressed through Kubernetes network policies [9], which enable communication between applications and services according to the given security policy.

Docker Compose Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file to describe *services* and *networks*, enabling the interaction of different containers. Docker Compose simplifies managing service dependencies, controlling networking, and scaling applications by running all components together in isolated yet interconnected environments. To support this third platform, our specialization template for this platform maps our abstract application scenario to Docker Compose services. Since the concept of communication channels does not exist in Docker Compose, we define separate networks. Each channel is mapped to a distinct network, with each communication partner assigned to the corresponding network [10]. This ensures that only applications intended to communicate with each other are able to do so.

6 Related Work

The scenario and policy description language is the core part of the work we present in this paper. Hence, we focus our discussion of related work on similar languages.

Markup Languages Markup languages offer a clear way to express and structure data, which is especially suitable for policy languages. Kubernetes, a prominent container orchestration platform, utilizes YAML to express network policies [9], regulating communication between applications and external endpoints. In contrast, XACML [11] is an XML-based language for run-time enforcement of Attribute-Based Access Control (ABAC) policies. Both solutions allow to filter rules, with complex rule expressions. However, Kubernetes YAML has a clear focus on network policies and is thus not directly applicable to our system configuration use case, where we also need to reason about attributes of individual programs. XACML can express a versatile range of policies, but this adds unneeded complexity for system configuration. Our Lua-based approach on the other hand, though more readable and easier to maintain, lacks the breadth of use cases that XACML supports.

Declarative Policy Languages Open Policy Agent (OPA) [12] employs the Rego language [13] to decouple decision-making from enforcement by evaluating JSON input data against rules. Another declarative language is Cedar [14], which is focused on fast evaluation of Role-Based Access Control (RBAC) and ABAC semantics. It is used, for example, in Amazon AWS Verified Access [15]. As declarative languages, they aim to simplify policy description and enhance readability. While OPA achieves this through logical "and" concatenation, our approach based on Lua offers flexibility in handling more complex statements within single rules. Cedar is tailored to run-time access control and thus may be an alternative for expressing policies. However, our work also targets application deployment and therefore requires additional flexibility, including support for imperative programming in the specialization templates. Our solution combines the flexibility of imperative programming with the ease of use of declarative descriptions. Specifically, system designers must describe applications, services, and policy rules in a declarative way in the form of nested dictionaries in the Lua language. However, unlike Cedar, our Lua-based approach currently does not allow formal verification.

[1] https://github.com/Barkhausen-Institut/M3

As another example from the class of declarative policy languages, Ponder [16] supports co-design and co-specification of both management and access control policies. It supports both run-time enforcement rules and structural declaration along with composition rules for their interdependencies. Ponder can support multiple stakeholders with possibly conflicting goals, but lacking ABAC semantics, it is not directly applicable to this work. Finally, DynaMo [17] is a language that allows to specify and formally analyze arbitrary access control policies based on a superset of ABAC semantics (DABAC[18]), but does not address the system configuration use case. Nevertheless, as soon as policy analysis receives more attention, DynaMo might become relevant in designing a more specialized Lua replacement in future work.

7 Future Work

We plan to continue our work in two major areas: (1) to support distributed systems with remote attestation, and (2) by employing formal methods.

TEEs and Remote Attestation Modern security platforms utilize Trusted Execution Environments (TEEs) to isolate applications from one another and, to some extent, from the underlying operating system. TEE-enabled platforms support remote attestation, allowing systems to provide verifiable evidence of the software running on them. Using our description language, remote systems can describe the applications and services running in their TEEs, along with the communication channels between them. Our enforcement engine, running on the verifying system, would validate this scenario description against a policy it considers acceptable. This description would complement the attestation evidence, which includes executable code measurements and TEE configurations.

Additionally, our system could be expanded to deploy multi-application scenarios across multiple systems in such a way, that it automatically creates cryptographically protected communication channels between remote TEEs. Remote attestation could also prove that these channels terminate in the correct TEEs.

Formalization of Policy Enforcement The second major area of future work focuses on formal methods to reason about (1) policy correctness, such as least-privilege compliance or prevention of unintended information flows [19], and (2) policy satisfiability. These properties need to be formally defined and embedded into our framework. As a side-product of such methods, we expect that a specialized declarative policy language will be required to replace Lua. Its underlying formal model should then unlock existing or new verification approaches (e.g. graph-based analysis of constraint dependencies [20] or workflow satisfiability analysis [21]).

8 Conclusion

In this paper, we presented a holistic approach that addresses both the automatic deployment of applications in distributed systems under a given security policy and the verification of third-party system trustworthiness. Our solution consists of two parts: (1) a description language, in which software vendors, system administrators, or users can describe what applications and services to run, along with policy rules, and (2) an enforcement engine that checks if the deployment scenario complies with the security policy. The description language is based on Lua, which is easy to write and understand, but also flexible enough to express arbitrary rules for policing communication and resource assignment for the target systems. Furthermore, the description language and the enforcement engine are platform independent. They can be adapted to a concrete target system via specialization templates also written in Lua. We demonstrated that capability by generating a working startup configuration for the M^3 microkernel platform and YAML-based configurations for Kubernetes and Docker Compose.

9 Acknowledgements

This research is funded by the European Union's Horizon Europe research and innovation program under grant agreement No. 101094218 (CYMEDSEC) and No. 101092598 (COREnext). It is also financed on the basis of the budget passed by the Saxon State Parliament in Germany.

References

[1] The Programming Language Lua. https://www.lua.org/. (Accessed: Oct 2024).
[2] Programming in Lua - Sec. 22.2: Other System Calls. https://www.lua.org/pil/22.2.html. (Accessed: Oct 2024).
[3] Programming in Lua - Ch. 21: The I/O Library. https://www.lua.org/pil/21.html. (Accessed: Oct 2024).
[4] Lua 5.3 Reference Manual - Sec. 6.10: The Debug Library. https://www.lua.org/manual/5.3/manual.html#6.10. (Accessed: Oct 2024).
[5] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 189–203. ACM, March 2016.
[6] Kubernetes. https://kubernetes.io/. (Accessed: Oct 2024).
[7] Docker compose. https://docs.docker.com/compose/. (Accessed: Oct 2024).
[8] Nils Asmussen, Sebastian Haas, Carsten Weinhold, Till Miemietz, and Michael Roitzsch. Efficient and Scalable Core Multiplexing with M^3v. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 452–466. ACM, February 2022.
[9] Kubernetes Documentation: Network Policies. https://kubernetes.io/docs/concepts/services-networking/network-policies/. (Accessed: Oct 2024).
[10] Overlay network driver. https://docs.docker.com/engine/network/drivers/overlay/. (Accessed: September 2024).
[11] OASIS Standard. eXtensible Access Control Markup Language (XACML) Version 3.0. https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html, January 2013.
[12] Open Policy Agent. https://www.openpolicyagent.org/. (Accessed: Oct 2024).
[13] Open Policy Agent: Policy Language. https://www.openpolicyagent.org/docs/latest/policy-language/. (Accessed: Oct 2024).
[14] Cedar Language. https://www.cedarpolicy.com/. (Accessed: Oct 2024).
[15] Secure Remote Access - AWS Verified Access - AWS. https://aws.amazon.com/verified-access/. (Accessed: Oct 2024).
[16] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. In Morris Sloman, Emil C. Lupu, and Jorge Lobo, editors, *Policies for Distributed Systems and Networks*, pages 18–38. Springer, 2001.
[17] Peter Amthor and Marius Schlegel. Towards Language Support for Model-based Security Policy Engineering. In Pierangela Samarati et al., editor, *17th International Conference on Security and Cryptography*, SECRYPT '20, pages 513–521. INSTICC, SciTePress, 2020.
[18] Marius Schlegel and Peter Amthor. Putting the Pieces Together: Model-Based Engineering Workflows for Attribute-Based Access Control Policies. In Pierangela Samarati et al., editor, *E-Business and Telecommunications*, volume 1795 of *Communications in Computer and Information Science (CCIS)*, pages 249–280. Springer Nature, 2023.
[19] Luigi Logrippo. Logical method for reasoning about access control and data flow control models. In Frédéric Cuppens et al., editor, *Foundations and Practice of Security*, pages 205–220. Springer, 2015.
[20] Peter Amthor and Martin Rabe. Command Dependencies in Heuristic Safety Analysis of Access Control Models. In Abdelmalek Benzekri et al., editor, *Foundations and Practice of Security (FPS '19)*, volume 12056 of *LNCS*, pages 207–224. Springer, 2020.
[21] Arif Khan and Philip Fong. Satisfiability and Feasibility in a Relationship-Based Workflow Authorization Model. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security – ESORICS 2012*, volume 7459 of *LNCS*, pages 109–126. Springer, 2012.