

Securing Workflows Using Microservices and Metagraphs

Loïc Miller, Pascal Mérindol, Antoine Gallais, Cristel Pelsser

▶ **To cite this version:**

Loïc Miller, Pascal Mérindol, Antoine Gallais, Cristel Pelsser. Securing Workflows Using Microservices and Metagraphs. Electronics, 2021, 10 (24), pp.3087. 10.3390/electronics10243087 . hal-03709704

HAL Id: hal-03709704
https://uphf.hal.science/hal-03709704

Submitted on 30 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

electronics

MDPI

Article

Securing Workflows Using Microservices and Metagraphs †

Loïc Miller [1,*], Pascal Mérindol [1], Antoine Gallais [2] and Cristel Pelsser [1]

[1] ICube, University of Strasbourg, 67081 Strasbourg, France; merindol@unistra.fr (P.M.); pelsser@unistra.fr (C.P.)
[2] LAMIH, CNRS, UMR 8201, Université Polytechnique Hauts-de-France, INSA Hauts-de-France, 59313 Valenciennes, France; antoine.gallais@uphf.fr
* Correspondence: loicmiller@unistra.fr
† This paper is an extended version of our paper published in IEEE 22nd International Conference on High-Performance Switching and Routing (HPSR 2021), Paris, France, 7–10 June 2021.

Abstract: Companies such as Netflix increasingly use the cloud to deploy their business processes. Those processes often involve partnerships with other companies, and can be modeled as workflows where the owner of the data at risk interacts with contractors to realize a sequence of tasks on the data to be secured. In this paper, we first show how those workflows can be deployed and enforced while preventing data exposure. Second, this paper provides a global framework to enable the verification of workflow policies. Following the principles of zero-trust, we develop an infrastructure using the isolation provided by a microservice architecture to enforce owner policy. We implement a workflow with our infrastructure in a publicly available proof of concept. This work allows us to verify that the specified policy is correctly enforced by testing the deployment for policy violations, and find the overhead cost of authorization to be reasonable for the benefits. In addition, this paper presents a way to verify policies using a suite of tools transforming and checking policies as metagraphs. It is evident from the results that our verification method is very efficient regarding the size of the policies. Overall, this infrastructure and the mechanisms that verify the policy is correctly enforced, and then correctly implemented, help us deploy workflows in the cloud securely.

Keywords: data leak; workflow; microservices; authorization; access control; policy verification; metagraphs; yawl; rego

check for updates

Citation: Miller, L.; Mérindol, P.; Gallais, A.; Pelsser, C. Securing Workflows Using Microservices and Metagraphs. *Electronics* **2021**, *10*, 3087. https://doi.org/10.3390/electronics10243087

Academic Editor: Vijayakumar Varadarajan

Received: 14 November 2021
Accepted: 8 December 2021
Published: 11 December 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Data leaks and breaches are increasingly happening. With more and more businesses using public clouds to process data [1], and with these data being frequently moved around, exposures are more likely to happen than ever. Those exposures are perceived as huge losses of money for businesses such as the movie industry [2], and as a loss of user privacy for applications dealing with user data [3].

To solve this issue, we aim to achieve a secure system enabling the exchange of data between non-trusted agents in the context of workflows. To this end, we combine two of our previous works [4,5] in this work. The first one describes an infrastructure to deploy a workflow securely in the cloud using microservices and how to verify the implemented policy is enforced, while the second details a way to verify the implementation of the policy corresponds to its specification.

1.1. Basic Concepts

There has been a steady increase in the number of breaches reported over the past eight years [6], 2019 being an all-time high with 5183 reported breaches as well as 7995 million records lost [6]. Malicious actors have been responsible for most incidents, but accidental exposure of data on the Internet (e.g., misconfigured databases, backups, end points, services) has put the most records at risk [6].

Even though both data breaches and data leaks result in data being exposed to unauthorized entities, the way these data are exposed is different. On one hand, data breaches refer to the unauthorized access of data by exploiting flaws in the security of the breached system. Data breaches can happen with data at rest [7] where attackers exploit a flaw to gain access to the data, or in transport [8], where attackers exploit a vulnerability to eavesdrop on traffic. On the other hand, data leaks refer to the exposure of data belonging to an entity, due to the way these data are processed by this entity, by a mistake [9], or caused by malicious behavior [10].

Preventing both forms of data exposure is a challenging problem to tackle, since an exposure can occur in multiple circumstances, caused by hacking, misconfigured databases, malicious insiders, etc. [3]. As attacks were generally assumed to come from a location external to the system via north-south traffic, the traditional way to secure such a system against those attacks was to protect it at the border via gateways, firewalls or programmable switches [11]. The recent rise of microservices as a paradigm, and their increased use in building large, cloud-based enterprise applications [12] has increased the attack surface, meaning protecting the network border is no longer sufficient. To prevent data leaks, one needs to consider attacks coming from inside the system (e.g., leaks stemming from the way data are processed or caused by a malicious employee). The zero-trust security model [13], where all traffic flows are required to be authenticated and authorized via fine-grained policies, provides such protection.

As per the zero-trust model, preventing agents from exposing data requires an authorization mechanism. Authorization is a key aspect of security, regulating the interactions taking place in a given system. For example, Netflix will often interact with their partners for some tasks, e.g., content ingestion [14,15]. Since the systems to be secured by authorization can be highly complex, administrators often rely on policy-based management of authorization. Policies define the desired behavior of a system from a high-level perspective. Hence, this form of management allows the separation of the problem of specification, i.e., defining the desired system behavior, from the problem of implementation, i.e., the enforcement of the desired system behavior.

Research on this topic mainly focuses on three areas: policy analysis, policy refinement and policy verification. On the one hand, policy analysis deals with the fulfillment of specific properties by a set of policies [16], e.g., detecting when two or more policies are conflicting. On the other hand, policy refinement handles the translation from high-level policies into low-level configurations [17].Depending on how this task is realized, the translation can lead to incorrect and/or non-optimized policy implementations; it can affect performance or even put the system at risk by introducing security flaws. According to the Verizon Data Breach Investigations Report, errors were causal events in 22% of data breaches [18]. As the risk of error increases when refinement is performed by hand, automatic or semi-automatic assisting tools have emerged to help administrators better translate their policies [19–21]. Lastly, policy verification is used to check whether the deployment of policies actually meets their high-level specification. Policy verification plays an important role since assisting tools are not free of errors, and deployment specificities can lead the policy to become erroneous. An erroneous policy can lead attackers to view files they were not authorized to see [22], access paid content free of charge [23] and even changing access rights [24] or deleting content [25].

We aim to perform this policy verification on workflows. We define a workflow as a sequence of tasks performed by a set of independent entities [26]. In practice, workflows are immensely important since they model business processes and define their interactions [27–29]. However, workflows can become complex and difficult to manage, especially in the case of multi-party workflows [30], which makes the problem of access control even more challenging. One of the most used languages to specify workflows is Yet Another Workflow Language (YAWL) [31].

1.2. Approach and Contributions

The data of the workflow should be secured at rest and in transport and cannot be exposed by any agent in both cases. To meet our requirements for zero-trust and prevent data leaks during the execution of workflows, we rely on a secured microservice architecture as well as a system to verify policies.

The microservice architecture allows us to design a system preventing data exposures that is simple, modular and scalable, thanks to its loosely coupled services. This is important when considering security mechanisms become challenging to configure, manage, scale and monitor when combined, with many actors using different IT environments.

We isolate in containers the environment in which the agents execute their tasks. Each container is secured thanks to a proxy enforcing access control. An orchestrator, a service mesh and policy engines are deployed to enforce the workflow along with the access policies of the owner.

We of course need to make sure those deployed policies are reliable. Therefore, this paper also presents a policy verification method, i.e., we check whether the deployment of policies actually meets their high-level specification.

We propose to model policies with a generic yet rich structure: metagraphs. We use its formal foundations to verify whether the actual deployment of a policy (i.e., its implementation) matches its initial specification.

We rely on this structure since, by design, it provides means to locate conflicts and avoid redundancy [32]. Metagraphs provide more accurate verification process than with other structures such as usual graphs. They belong to the rare appropriate structures able to naturally model access control policies. Their formal foundations enable us to perform fine-grained verification on possibly large policies. Transformations such as projections or context metagraphs [33] can be used to help with the visualization of very large policies.

This work is merges and extends two of our previous papers [4,5] published in the IEEE 22nd International Conference on High-Performance Switching and Routing (HPSR 2021) conference. The contributions of our paper are as follows:

1. First, we develop an infrastructure using the isolation provided by microservices to enforce policy;
2. We implement a workflow with our infrastructure in a publicly available proof of concept and verify that our implementation of the specified policy is correctly enforced by testing the deployment for access control violations;
3. We measure performance of our infrastructure with policy engines and find the overhead cost of authorization to be reasonable for the benefits;
4. Second, we rely on metagraphs to perform the verification of access control policies (to the best of our knowledge we are the first to do so).We argue they represent one of the most appropriate form of policy modeling enabling refinement and verification to finely pinpoint implementation errors;
5. We then propose a suite of translation tools enabling policy verification; More specifically, we introduce how to perform such verification on a workflow-like policy specification. We rely on a policy implementation based on Rego, a high-level declarative language built for expressing complex policies;
6. Finally, we conduct a thorough performance evaluation of this second contribution. We verify that deployed policies match their specification in a very reasonable time, even for large workflows with a substantial number of rules.

Considering the problem at hand (Section 3), we specify our threat and security models. We describe our solution (Section 4) and its companion proof of concept (Section 5), then evaluate the authorization overhead (Section 6). For our second part on policy verification, we provide some background on metagraphs and describe our verification procedure (Section 7), then evaluate its performance (Section 8). We finally review related works (Section 2) and conclude (Section 9).

2. Related Works

Existing works provide guidance on overall security requirements and strategies for microservices [34], as well as guidance on more specific microservices components such as service mesh [12,35] or containers [36,37]. Chandramouli [34] provides guidance on security strategies for implementing core features of microservices, as well as countermeasures for microservices-specific threats. We follow the guidelines and recommendations presented in these works. Contrary to those works, we propose a complete infrastructure, accompanied by a real-world deployment, as well as both a security and a performance evaluation of this deployment.

Weever et al. [38] investigate operational control requirements for zero-trust network security, and then implement zero-trust security in a microservice environment to protect and regulate traffic between microservices. They focus on implementing deep visibility in the service mesh, and do not propose a security or a performance evaluation. Hussain et al. [39] propose and implement a security framework for the creation of a secure API service mesh using Istio and Kubernetes. They then use a machine learning-based model to automatically associate new APIs to already existing categories of service mesh. Contrary to our work, they use a central enterprise authorization server, in opposition to our policy sidecars. Zaheer et al. [40] propose eZTrust, a policy-driven parameterization access control system for containerized microservices environments. They leverage eBPF to apply per-packet tagging depending on the security context, and then use those tags to enforce policy, in opposition to our enforcement of policy which relies on policy sidecars local to the services.

On the side of formal analysis of data leaks in workflows, Accorsi and Wonnemann [41] proposed a framework for the automated detection of leaks based on static flow analysis by transforming workflows into Petri nets. Some papers propose data leak protection, by screening data and comparing fingerprints [42–48]. Segarra et al. [49] propose an architecture to securely stream medical data using Trusted Execution Environments, while Zuo et al. investigate data leakage in mobile applications interaction with the cloud [50].

There exist several pieces of works on policy analysis, refinement and verification in the literature. Policy analysis mainly deals with policy evaluation and anomaly analysis: checking for errors such as incorrect policy specifications, conflicts and sub-optimizations affecting either a single policy or a set of policies [16] being the primary research topic. Works in this area use different techniques to achieve this goal, such as model checking [51,52], binary decision diagrams [53], graph theory [54], Deterministic Finite State Automata (DFSA) [55], First Order Logic (FOL) [56], geometrical models [57], answer set programming [58], petri nets [59] and metagraphs [32].

Policy evaluation instead deals with checking whether a request is satisfied by a set of policies. It is typically used to verify the effective impact of modifying a policy. Works that deal with analyzing the impact of changes in a policy usually model those policies and then analyze the obtained representation for effective impact [60,61].

Policy verification, the subject of this paper, deals with checking whether a policy is correctly enforced in a system. There exists only a few works on policy verification [62,63], when compared to the large body of work dealing with policy analysis, and none of them uses metagraphs. On the one hand, Hughes and Bultan [62] as well as Bera et al. [63] propose automatic verification of access control policies against a set of properties. Verification is achieved by translating the properties into a Boolean satisfiability problem and using a SAT solver, whereas we use metagraphs which come with a useful visual representation of the policies.

On the other hand, even though metagraphs have emerged as one of the most suited tool for representing and reasoning about policies, they are still underused with only a few existing works in the literature [32,33,64–66]. Basu and Blanning [33] compiled all the research on metagraphs up until 2007 in a book, which is the reference for general metagraph theory and applications.

Ranathunga et al. [64] defined a toolkit in python to manipulate metagraphs. Hamza et al. [65,66] used metagraphs to model policies in IoT devices to generate and validate Manufacture Usage Descriptions (MUD) profiles—it can be used to define the access control model and network functionality these devices need to properly function.They also check compliance of those MUD profiles with different levels of security policies, to determine where those devices are safe to be deployed. Closer to our contribution, Ranathunga et al. [32] use metagraphs to model network policies for distributed firewalls. In particular, they use specific metagraph properties to detect redundancies and conflicts in those policies. Contrary to our work, they do not verify deployed policies against specifications.

3. Threat and Security Model

We define a workflow as a sequence of tasks to be performed by a set of independent actors. The owner of the data (i.e., the instigator of the workflow) interacts with contractors to realize such a sequence. Both the owner and the contractor have agents processing the data, where agents can represent an employee or an automatic service.

Let us consider a simple example of workflow (Figure 1), where an owner in the post-production stage of making a movie employs other companies to edit the video and audio components. This example is inspired by the concepts presented by Byers et al. [2], but simplified for the sake of the example. The owner (O) first sends its data to the company responsible for special effects (C_{1_x}). C_{1_x} applies special effects to the movie sequences the owner sent him, and then sends the result to the company responsible for coloring (C_2) as well as to another for sound mastering (C_3). C_2 then ships its result to the agent in charge of High Dynamic Range (HDR) (C_3) and sound mastering (C_4). Finally, both C_3 and C_4 sends their output back to the owner.

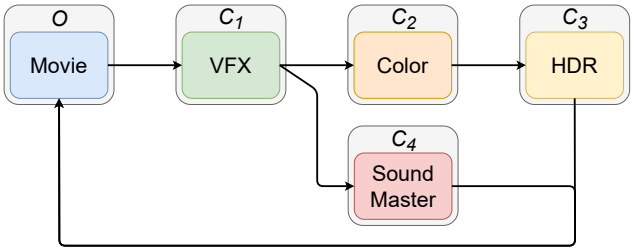


Figure 1. Movie workflow example. Arrows model the specified workflow, and thus represent the communication flow of our example. The owner (O) sends its data to the first contractor (C_1), for special effects processing. C_1 then sends the modified data along the workflow, for color (C_2), HDR (C_3) and sound (C_4) processing. The resulting data are then sent back to the owner.

We consider a threat model from the point of view of each actor. The owner wants to avoid the leakage of the data sent to the involved contractors. The threat here is then that an agent leaks the critical data to an unauthorized party (or that the data are accessed by an adversary). On the other hand, a contractor does not want the other actors, including the owner, to learn about their business intelligence.

3.1. Trust Model—Actors and Environment

From the point of view of the data owner, trusting the contractors is one thing, trusting its agents another. In other words, if the owner trusts the organization of the contractor to not intently bypass our system, controlling the actions of the contractor's agents is then possible for both the owner and the contractors. If one does not trust the contractors to deploy the infrastructure they are required to deploy, there is no easy way to verify that the data are actually sent to the secure environment we designed (Section 4), therefore removing any guarantee we might have concerning data leaks.

In contrast, looking at a finer granularity, actors do not need to trust their agents and the ones of the other actors. Even though agents are deterred from engaging in

malicious activities, due to the nature of their relationship with their companies (internal rules, non-disclosure agreements, ...), they can still put data and/or business intelligence at risk through accidental exposure or malicious behavior. Actors are thus assumed to be malicious. Our solution controls those agents to prevent owner data and business intelligence leaks. This is consistent with our need to trust the contractors, since business to business contracts have the same deterrents, but with much higher stakes at play.

From the point of view of a contractor, we have the same trust issues as the owner. Other actors, including the owner, might try to reverse engineer the business intelligence of the contractor. This reverse-engineering process requires a lot more effort than simply having access to the data of the owner, and might prove to be very hard or even impossible to do in some cases. This can happen in very specific cases, such as when a contractor receives its input(s) and gives its output(s) to the same actor. As data are encrypted in transport, only the two ends of a communication see the data. A solution would be to insert the owner between contractors such as to limit their learning of the workflow and trust the owner not to reverse engineer the actions of its contractors. In the same way the owner needs to trust that contractors do not intently bypass our system, the contractors need to trust that actors sending them data do not tamper with it. Like the owner, contractors do not trust the agents.

Finally, both the owner and the contractors need to trust the owners of the environments involved in the workflow. Although the environment an actor is using can be owned by this actor, meaning the added trust requirement is the same as trusting the actor, some actors can use a third-party environment to fulfill their task(s) (e.g., a cloud provider). Since this third-party provides (part of) the environment the workflow will be deployed on and has admin rights to the machines supporting the workloads, it can try to gain access to the data of the owners or the business intelligence of the contractors. We would need to enhance our solution with Trusted Execution Environments (TEEs) to fully remove the need for trust in those third parties. With the proposed infrastructure, one needs to trust those potential third parties. As such, actors and environment providers are considered honest but curious.

To summarize, from the point of view of the owner or a contractor, we trust everything but the agents. Actors are assumed to be honest but curious, while agents are assumed to be malicious.

3.2. Attacker Model—External Attackers and Malicious Agents

Taking into account the assets to protect and our trust model, we consider three types of attackers in our model.

- *External attacker*: External to the workflow and the location of the deployed infrastructure. Such attackers try to gain access to the data or the business intelligence from the outside.
- *Co-located attacker*: External to the workflow, but co-located at the deployment (e.g., an attacker located in one of the third-party clouds). This co-located position opens more exploit options.
- *Malicious agent*: Internal to the workflow, this attacker tries to leak the data outside.

Despite the fact our model already covers most cases, it does not deal with the full range of possible attacks. Fully protecting against some attacks (e.g., from a contractor or a third-party cloud provider) would make the system less convenient and usable for contractors or the owner. Protection from leaks resulting from physical attacks such as when an employee takes a picture of his screen are not considered.

4. Infrastructure and Proof of Concept

We now present the infrastructure we propose for protecting a workflow execution from the threats expressed in Section 3. As we need a way to prevent data leaks, we need to control the communications an agent can engage in. To achieve this, we need to control the environments the agents will be using, to make sure that all the actions of an agent follow

a policy enforced by the owner. We opted to do this using the microservice architecture, for the benefits granted by the components of the infrastructure listed below. Moreover, they are already commonly deployed to provide many services such as automatic scaling and isolation.

Overall Description of the Infrastructure

In this infrastructure, agents of our workflow are mapped to containers, which are then used in conjunction with an orchestrator, a service mesh and policy engines to enforce the policy of the owner.

Figure 2 shows the workflow we defined in Figure 1, with each actor having its own deployment space represented by the cloud surrounding the boxes which represent the agents of those actors (e.g., the C_{1_1} box represents an agent of contractor C_1). The access policies of a service are pushed in the policy sidecar associated with the service.

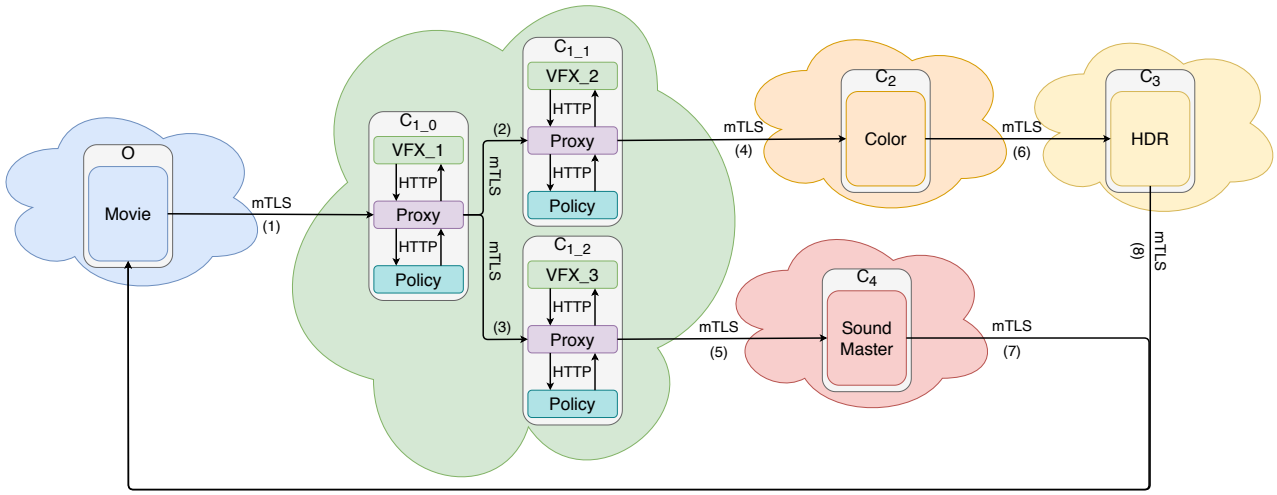


Figure 2. Secure infrastructure. Each box represents an agent. It is a pod with the appropriate containers. The container of the color of the actor represents the service. Purple containers represent the proxies of the service mesh, and blue containers represent the policy sidecars. The arrows stipulate whether the communications are secure (`mTLS`) or not (HTTP).

Figure 2 also illustrates how we use the elements of the microservice architecture. Each agent is a pod, containing the service (i.e., the environment the agent will be using), a proxy and a policy sidecar. The proxy sidecar will intercept all traffic coming from and going to its respective service. The proxy will then check thanks to the policy sidecar if the request is authorized or not. If the request is authorized, it is forwarded accordingly, and the request is rejected otherwise. Proxies are configured by the service mesh controller (Figure 3), providing them with identities and key pairs, as well as routing information for them to initiate secure communications with other proxies in the mesh. Policy is pulled periodically by the policy sidecars from a policy store, which allows for policy changes. Since the service mesh controller and the policy store are under the control of the owner, he is in control of the system. Thus, the owner specifies the policy to be applied to enforce the desired workflow, preventing data from leaking outside.

The data processed by the pods is stored on mounted Persistent Volumes (PVs), which are encrypted with a key located in a key-value store of the orchestrator, providing us with **data security at rest**. We generate a key to encrypt each PV required by needs of the workflow. Since the keys are all stored in the same key-value store, this does not really mitigate risks against a technically skilled attacker gaining access to the key-value store, but it can help to protect some of the data in case the attacker only gains access to a subset of the keys through other means. Since the keys are stored in the master components of the orchestrator, they are under the control of the owner. To enforce the workflow and make sure the agents cannot bypass it via the PVs, each agent must have its own personal PV.

Pods can also communicate according to the specified workflow and policy via `mTLS`, providing us with **data security in transport** as indicated by the communications between the pods in Figure 2. Communications inside a pod are not encrypted, but the isolation layers protect the data against eavesdroppers.

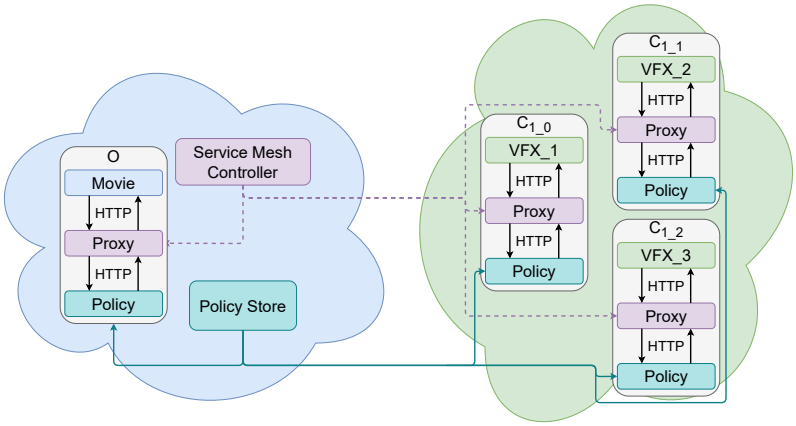


Figure 3. Representative subset of the secure infrastructure control plane (contractors C_2 through C_4 are not represented). Proxies are configured by the service mesh controller, providing them with identities and key pairs, as well as routing information for them to initiate secure communications with other proxies in the mesh. Policy changes are enabled with periodical pull on the policy sidecars (whose input comes from a policy store).

Once a service has completed all the tasks he was assigned to do, the associated pod is destroyed to make sure data cannot be leaked from this service past this point in time. To provide data security in transport, services in the service mesh are provided with an identity in the form of a certificate, which is associated with a key pair. To make sure those are safe to use, and that no attacker gained access to the keys or tampered with them before they reach the appropriate service, we need to verify the key distribution process is secure. In the case of the service mesh, this is done automatically for us.

Thanks to this infrastructure, communications can be constrained to follow a policy, giving us a streamlined way to prevent data leaks. We show how a simple policy to prevent data leaks can be defined.

5. Proof of Concept

We realized a proof of concept, by implementing the infrastructure described in Section 4. We reproduce the workflow of Figure 2, with services of the workflow receiving and sending arbitrary data to represent the data of the owner. We use Docker [67] for our containers, Kubernetes [68] for our orchestration layer, Istio [69] as our service mesh, using Envoy [70] for the proxy sidecars and Open Policy Agent (OPA) [71] for the policy sidecars. We also use Kubernetes to provide the services with encrypted volumes. This infrastructure was deployed on Google Cloud Platform (GCP), using one cluster for each actor of the workflow, for a total of five clusters. Each cluster runs one *n1-standard-2* node (2 vCPUs, 7.5 GB of memory), on version *1.14.10-gke.36*, except the cluster of the owner which runs two of them, since running the control plane requires additional resources. The clusters for the owner, color and VFX are in us-central1-f whereas the clusters for HDR and sound are in us-west2-b.

The workflow we want to enforce is shown in Table 1, where each row represents the *source* of a request, and each column a *destination*. The agents can also send GET requests, but they are all denied by the policy.

The complete data, code as well as guidance to realize this Proof of Concept are publicly available (See github.com/loicmiller/secure-workflow, accessed on 10 December 2021).

We also developed a test framework to check that:

- Traffic is either encrypted or protected inside a pod by the isolation provided by the pods;
- The policy, allowing or denying communications between services, is correctly enforced.

Table 1. Proof of Concept policy.

Source	**Destination**						
	Owner	$\boldsymbol{VFX_1}$	$\boldsymbol{VFX_2}$	$\boldsymbol{VFX_3}$	**Color**	**Sound**	**HDR**
Owner	-	POST					
$\boldsymbol{VFX_1}$		-	POST	POST			
$\boldsymbol{VFX_2}$			-		POST		
$\boldsymbol{VFX_3}$				-		POST	
Color					-		POST
Sound	POST					-	
HDR	POST						-

To do so, we capture traffic on every network interface in the service mesh and perform each possible communication. In the general case, considering we have N services and M types of request, we obtain the number of possible communications with the formula: $N(N-1)M$. Since we capture on each interface, and services have a loopback as well as an external interface, we obtain the total number of required captures: $N(N-1)M(2N) = 2(N^3 - N^2)M$. The number of required captures thus grows cubically with the number of services and linearly with the number of requests.

Considering our previous example in Section 5, we have seven services, two possible requests (GET and POST), which gives us a total of 1176 captures. Captures are obtained from a `tcpdump` container added to the service pods as a sidecar. Since containers in the same pod share the same network namespace, capturing traffic from the `tcpdump` container on either the loopback or the external interface allows us to see traffic from the other containers in the pod. Figure 4 shows the path a communication takes inside the service mesh, as well as whether traffic is encrypted or not. The request is initiated by service A, and intercepted by its associated proxy via the loopback interface. The proxy will then check thanks to the policy sidecar if the request is authorized or not. If the request is authorized, it is forwarded accordingly. The request is rejected otherwise. In the case where the request is authorized, it is forwarded to the proxy of service B using `mTLS`. There, the proxy forwards the request to service B, which replies by going through the same steps as earlier. Traffic going to/coming from the loopback should be unencrypted, whereas traffic going to/coming from the external interface should be encrypted. Our captures show that this is indeed the case. Traffic does not need to be encrypted on the loopbacks, as all the elements (i.e., the service and its sidecars) that have access to this loopback are in the same trust zone. The layers of isolation provided by the pods protect the loopback traffic from being seen by unauthorized entities.

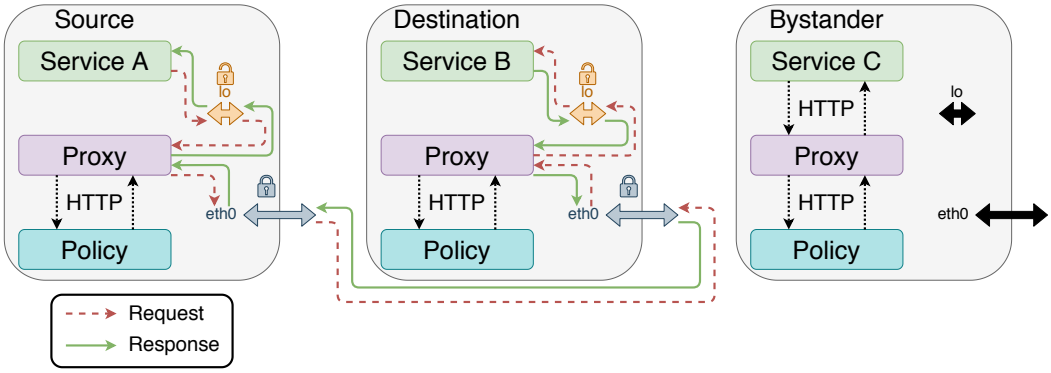


Figure 4. Detailed view of pods and the communication flow. Traffic is unencrypted on the loopbacks, but encrypted on the external interfaces.

Obviously, pods in the service mesh have one of three roles during a communication. Either they are the source of the communication, the destination of the communication, or simply a bystander that is not involved in the communication. This is important, because the checks we need to perform depend on where traffic was captured:

- **Source/Destination** loopback: We need to verify that a communication between the source and the destination is occurring (i.e., correct IP addresses and ports). We need to verify that the request in the capture corresponds to the request we are testing for (GET or POST). The response needs to be in accordance with the policy: in this case, '403 Forbidden' if the policy was 'deny' and '200 OK' (GET) or '201 OK' (POST) if the policy was 'allow'.
- **Source/Destination external interface**: We need to verify that a communication between the source and the destination is occurring (correct IP addresses and ports). We need to verify that the traffic is encrypted by `mTLS`, and not passed in clear text.
- **Bystander loopback and external interface**: We need to verify that no communication between the source and the destination is occurring, whether encrypted or unencrypted.

We built a tool that automatically extracts the authorization policies from the OPA policy configuration, generates and then tests an access control matrix. For each possible communication in the service mesh, our tool loads all the captures relevant to this communication, identifies them to see what we should verify in each capture, and then proceeds to check if captures are in accordance with the criteria above. It is then easy to evaluate whether the system is compliant with the overall policy. The complete code for the test framework is publicly available (See github.com/loicmiller/secure-workflow, accessed on 10 December 2021).

6. The Overhead of Security

In this section, we analyze the performance overhead added by the policy sidecar enforcing security. We measure the pod startup time and the request duration (between each couple of connected pods).

Startup time

we first evaluate the impact of having an additional container for OPA on the startup time of pods. An independent-samples t-test was conducted to compare startup times in a deployment of our PoC with or without OPA. We gathered 130 observations per pod and per deployment ($N = 1820$ in total).

Figure 5 allows measurement of the cost on the initial deployment by comparing the distribution of startup times (with or without OPA deployed). The group with the OPA sidecar exhibits significantly higher startup times compared to the group without the OPA sidecar, $t(1818) = 43.19$, $p < 0.001$. Pods with OPA have a substantial increase in startup time of almost two seconds on average, i.e., 32.72% of the startup time. More in details, the *effect size* for this analysis, $d = 1.985$, was found to exceed Cohen's convention for a

large effect ($d = 0.80$). Running a *post hoc power analysis* also reveals a high statistical power, $1 - \beta > 0.999$.

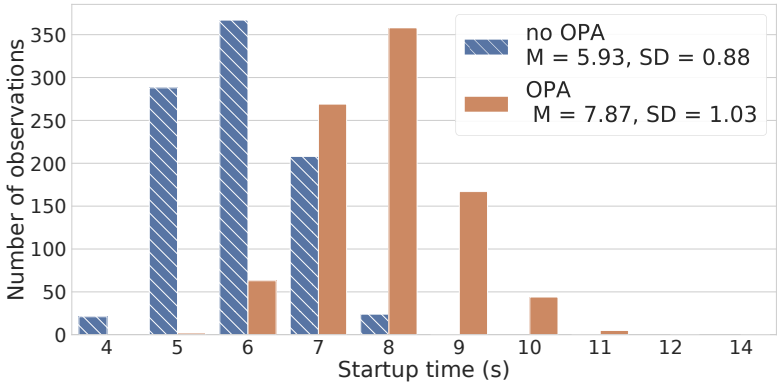


Figure 5. Distribution of startup time in deployments with/without OPA.

Request time

To test whether the policy is scalable for more complex workflows, we measure the influence of policy size on communications. A *one-way between subjects ANOVA* was conducted for each type of communication (intra/inter-region) to compare the effect of policy size on request duration in five increasing orders of policy size: `no opa`, `all allow`, `minimal`, `+100` (rules) and `+1000` (rules).

The `no opa` policy deployment corresponds to having no OPA container at all. The `all allow` policy deployment corresponds to having no rules and allowing all communications by default. The `minimal` policy deployment corresponds to having the default minimal number of rules to enforce the workflow of the PoC. The `+100` and `+1000` correspond to the minimal policy being inflated respectively with 100 additional rules (+147%) and 1000 additional rules (+1470%), with additional rules being obligatorily evaluated by OPA.

For each ANOVA, we gathered 40 observations per authorized communication per level of policy ($N = 1600$ in total). Figure 6 shows the distribution of request duration for each policy size. For intra-region communications, there is a significant difference in request duration among the five scenarios of policy deployments, $F(4, 795) = 364.05$, $p < 0.001$, $\eta^2_p = 0.65$. For inter-region communications there also exists a significant difference (in request duration) among the five scenarios of policy deployments, albeit with a lesser effect: $F(4, 795) = 15.23$, $p < 0.001$, $\eta^2_p = 0.07$. See https://github.com/loicmiller/secure-workflow, accessed on 10 December 2021, for full data, code and statistical analysis in the form of jupyter notebooks.

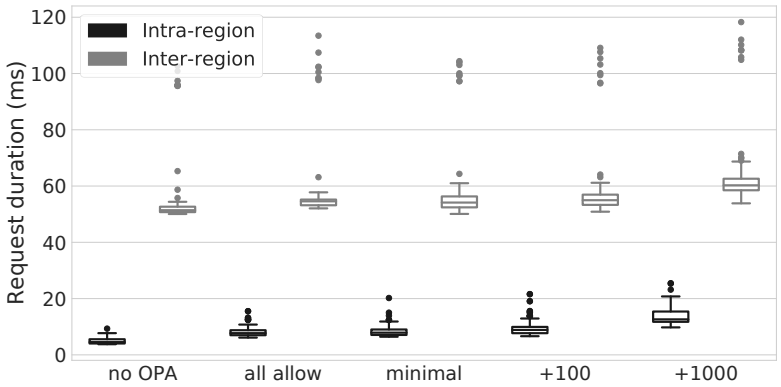


Figure 6. Spread of request duration in intra and inter-region communications by policy size.

Although our results suggest that a higher policy size increases request duration, it should be noted that this size should be strongly increased to observe an effect. This effect

is minor in inter-region communications. In summary, the added security provided by the workflow enforcement costs pods two seconds of startup time on average, and either 7% or 65% of the variance in request duration.

7. Verify the Deployment of the Access Control Policy Using Metagraphs

Having described our underlying technical infrastructure, we now tackle the issue of checking whether the deployment of policies actually meets their high-level specification. For this purpose, we rely on metagraphs that efficiently and finely model access control policies. As an added benefit, metagraphs can be used to detect redundancies and solve conflicts in such policies. We first provide a bit of background on metagraphs, which enable us to perform our fine-grained verification.

7.1. Background: An Expressive Model

A metagraph is a generalized graph theoretic structure such as directed hypergraphs, which is defined as a collection of directed set-to-set mappings. Each set (containing subsets or elements) in the metagraph is a vertex, and directed edges represent the relationship between sets. More formally, a metagraph can be defined as follows:

Definition 1 (Metagraph)**.** *A metagraph* $S = \langle X, E \rangle$ *is a graphical construct specified by a generating set* X *and a set of edges* E *defined on the generating set. A generating set is a set of elements* $X = \{x_1, x_2, ..., x_n\}$*, which represent variables of interest. An edge* e *is a pair* $e = \langle V_e, W_e \rangle \in E$ *consisting of two sets, an invertex* $V_e \subset X$ *and an outvertex* $W_e \subset X$*.*

Figure 7 illustrates a conditional metagraph. Conditional metagraphs are metagraphs augmented by propositions, i.e., statements that can either be true or false. A proposition attached to an edge must be true for the edge to be used in a path. Each edge may contain zero or more propositions and each proposition may be used in multiple edges. Overall, Figure 7 represents the necessary tasks for employees to perform a bank transfer. Edges (e_1, e_2, e_3) relate sets of employees (u_1, u_2) and tasks (*create_form*, *fill_form*, *review_form*, *transfer_money*). They contain an arbitrary number of propositions, e.g., $tenure > 2$ for e_1. Using an edge depends on the evaluation of its propositions, e.g., both employees can perform the operations *create_form* and *fill_form* via e_1 provided they have more than two years of experience. In Figure 8, we model the workflow of Figure 1 with added constraints as a conditional metagraph.

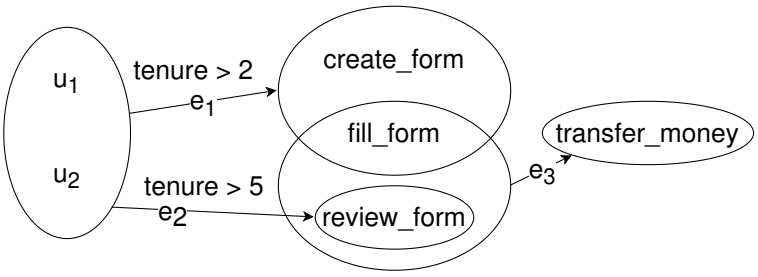


Figure 7. A simple example of conditional metagraph to model the following question: what are the necessary tasks for employees to perform a bank transfer?

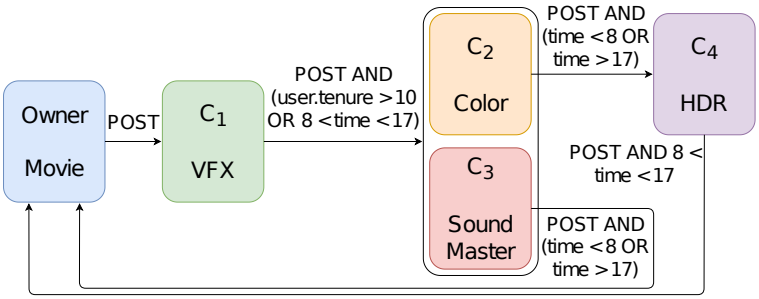


Figure 8. Movie workflow: special effects apply before color tuning and sound mastering. HDR is set up last.

The notion of simple path, i.e., a sequence of edges $\langle e_1, e_2, ..., e_n \rangle$ from an element x to an element y where $x \in invertex(e_1)$, $y \in outvertex(e_n)$ and for all e_i, $i = 1, ..., n-1$, $outvertex(e_i) \cap invertex(e_{i+1}) \neq \varnothing$, does not describe all the connectivity properties existing in a metagraph. For example, in Figure 7, there are two simple paths from $\{u_1, u_2\}$ to $\{transfer_money\}$, (e_1, e_3) and (e_2, e_3). However, none of them can perform $transfer_money$ as they respectively do not reach either $review_form$ or $fill_form$, which are both necessary to perform $transfer_money$. Using the set consisting of all three edges (e_1, e_2, e_3) is necessary (and sufficient) to perform $transfer_money$, but it is not a simple path: there does not exist a simple sequence of edges consisting of these three. Such a set of edges $\langle e_1, e_2, e_3 \rangle$ is called a metapath [33].

Reachability between the source and target sets in a metagraph is defined by the existence of (valid) metapaths between the two. Additionally, metapaths have a dominance property which can be used to determine redundant components (edges or elements) [32]. Once identified, those components can be safely removed from the policies. A metapath is *input-dominant* if no proper subset of its source is also a metapath to its target, *edge-dominant* if no proper subset of its edges is also a metapath to its target, and *dominant* if it is both input-dominant and edge-dominant [33].

7.2. Comparing the Specification with Its Implementation

By modeling the high-level policy specification as well as the translated policy implementation as two metagraphs, we can compare both to track (distributed) deployment errors. When specification and implementation metagraphs match, the policy implementation has been correctly translated from the policy specification. If they do not match, the metagraphs are not equivalent: errors occurred during the refinement and/or deployment. Figure 9 summarizes our approach.

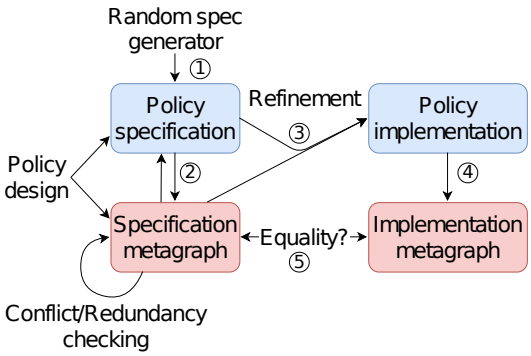


Figure 9. Enabling policy verification using metagraphs. We propose 4 tools: ① RandomWorkflowSpecGenerator; ② ③ SpecToRego; ④ RegoToMetagraph; ⑤ SpecImplEquivalence.

Modeling Workflows and Their Policies

For our purposes and evaluations, we consider the verification of policies enforcing workflows. In Figure 8, propositions on the edges constrain the communications. For exam-

ple, C_1 can only send data to C_2 and C_3 if the communication is a `POST` request, and either the tenure of the user is greater than 10, or the request happens between 8 AM and 5 PM.

In practice, we consider Yet Another Workflow Language (YAWL) [31] as the default language for specifying workflows. Considering YAWL [31] as the most representative tool to specify, analyze and execute workflows today, we will show how to transform any of its modeling components (e.g., composite tasks, conditions) into a metagraph representation.

YAWL nets

A workflow specification in YAWL is a set of extended workflow nets which form a hierarchy [31]. Each of these nets represents a (sub-)process. The net at the top of the hierarchy is called the root net. Tasks in a process can be either atomic or composite, where composite tasks represent another net, but at one lower level in the hierarchy. This separation in nets is useful when the process becomes too complex to manage, and can be broken down in smaller pieces. For example, a process for an online store might comprise a `checkout` task, but this task can be comprised of multiple tasks such as `choose shipping` or `choose payment mode`. To model this example in YAWL, the `checkout` task would be the composite task in the root net, while the other tasks are part of the `checkout` net at one lower level in the hierarchy.

In a metagraph, we can model tasks as edges of the metagraph, where the inputs (outputs) of a task correspond to the invertex (outvertex) of the edge. Using metagraphs, we can model atomic tasks as edges, and composite tasks can be edges representing the composite tasks, with each composite task being its own sub-metagraph, mirroring the nets in YAWL.

Processes and workflows

A process may contain information elements which are not yet evaluated. A workflow is an instantiation of a process for a set of particular values. Thus, a process can result in multiple workflows. Taking into account this definition, a process can be modeled by a conditional metagraph, with all its propositions still not evaluated. A metagraph with no propositions, i.e., a simple metagraph, can represent a workflow, i.e., one particular instantiation of the process [33].

Conditions and operators

Each net has one input condition and one output condition, which represent the start and end points of the process. In addition, there is the possibility to use operators, namely `AND`, `OR`, and `XOR`, which control the flow of information between tasks of the process. Each of these operators have a split and a join variation, which indicate the behavior of the operator. For example, an `AND-split` indicates the workflow executes all branches of the split, whereas an `AND-join` indicates the tasks reaching the join must all be completed for the next task to be executed. YAWL also uses conditions (besides the start and end), which represent a state the workflow is in after finishing a task, but before starting a new one. Those conditions are useful when users make some decisions on their own while the workflow system cannot pre-determine their choices.

For example, let us consider the film production process represented in Figure 10. This case study focuses on the film production process, representing the chain of information processing realized along the shooting of the movie. This process taken from the YAWL foundation website [29] was realized in collaboration with the Australian Film Television and Radio School (AFTRS), with the help of domain experts.

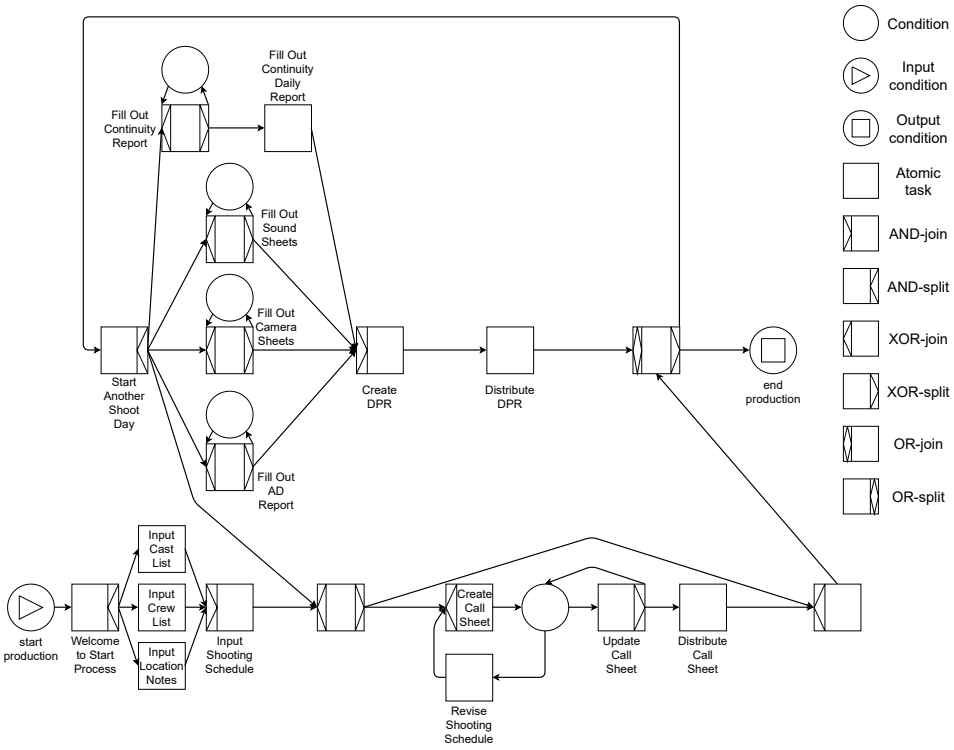


Figure 10. Film production process represented in YAWL. This case study represents the chain of information processing realized along the shooting of the movie. Although the shooting is taking place during the day, a designated crew collects the information associated with the shooting via production forms, which are used to produce the daily progress report. A call sheet containing all the information concerning logistics and necessities is also created.

A condition is used with the `Fill Out Sound Sheets` task, indicating the need for this information to be checked by the production manager before proceeding to task `Create DPR` [29]. Those modeling elements are all represented in Figure 10.

7.3. Roadblocks of Our Model

In YAWL, a task can have any input and any output, or can even have no input or output at all. At first glance, this seems to limit our model in two ways. First, since we model tasks as edges, tasks with no input (output) have no invertex (outvertex). To overcome this limitation, we add placeholder variables to empty vertices, which have no effect, but extend our modeling abilities to include such tasks. Conditions are also modeled using placeholder variables since they do not have any input or output.

Second, in YAWL, two tasks taking place sequentially in the process are not necessarily linked by their variables. In other words, the outputs of the first task do not necessarily correspond exactly to the inputs of the second task. This may look troublesome, since our metagraph representation relies on those links to retain the information of the workflow (i.e., which task should be performed next?), and makes use of paths and metapaths in its analysis. To enforce the sequence of tasks, we need to add propositions to the metagraph in certain cases.

There are two possibilities when considering two tasks chained sequentially, regarding the outputs of the first, and the inputs of the second. The first possibility is that the outputs of the first task correspond at least partially to the inputs of the second task, i.e., at least one variable is common to the two sets. In this case, no propositions need to be added since we have at least one shared variable, modeling the link between those two tasks. The second possibility is that there is no correspondence between the outputs of the first task and the inputs of the second task—the tasks are disjoint. For example, this occurs when the outputs of the first task are not necessary to perform the second task, but we still

want the tasks to be executed in this specific order, such as the `Input Shooting Schedule` task of Figure 10. In this case, we need to create a new edge from the outputs of the first task to a proposition signifying the first task has been completed. Once this is done, we can add this proposition to the input of the second task, to preserve the link between the tasks.This is shown in Figure 11, where the task `Input Cast List` does not share variables with its next task, `Input Shooting Schedule`. If the proposition is true, the task has been completed and the second task can start—the task becomes unavailable otherwise. Thus, the proposition clarifies the fact that `Input Cast List` needs to be completed to proceed to the `Input Shooting Schedule` task.

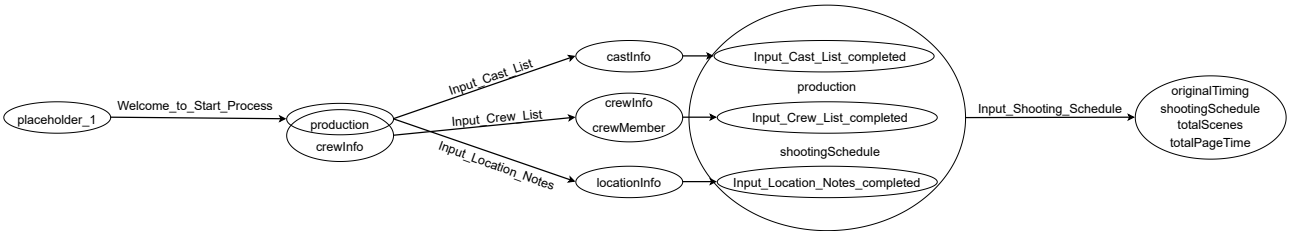


Figure 11. Film production process represented with a metagraph. Only the start of the YAWL process is represented for brevity. The task `Input Cast List` does not share variables with its next task, `Input Shooting Schedule`, thus we add a proposition indicating the completion of the task.

The `AND`-split, `OR`-split and `XOR`-split elements are handled naturally in a metagraph. Since each task is represented by an edge, a split simply corresponds to multiple edges sharing the same invertex. However, modeling the `AND`-join, `OR`-join and `XOR`-join elements is a bit more complex.

With join operations, we cannot say it simply corresponds to multiple edges which share the same outvertex, since, depending on the operator, we might want all or only some of the tasks preceding the join to have been executed. The `OR`-join requires at least one of the preceding tasks is executed, so no additional measures need to be taken. An `AND`-join on the other hand requires all tasks preceding the join to be executed, whereas a `XOR`-join requires only one of them is. To model this intent, we add completion propositions to the metagraph, in the same way we did for disjoint tasks. Those propositions are illustrated in Figure 11, where the tasks `Input Cast List`, `Input Crew List` and `Input Location Notes` are AND-joined in the next task, `Input Shooting Schedule`. It follows that the `AND`-join needs all propositions to be true, whereas the `XOR`-join needs exactly one of them to be true. To check for the fact all other propositions are false in the case of the `XOR`-join, we create non-completion propositions in addition to completion propositions. A non-completion proposition is true if the task has not been executed, and false otherwise.

Using this metagraph representation brings us many advantages. First, it is easier to identify and analyze workflows associated with a process, via the evaluation of the propositions in the conditional metagraph representing the process. This representation can also help us analyze the independence of decomposed subprocesses, as well as the redundancy and full connectivity of composite processes via the union of metagraphs [33]. We can also identify more easily interactions between/among informational elements and/or tasks. For example, we can simply analyze how do informational elements relate to each other through tasks, how do tasks relate to each other, and even which tasks might be disabled if a resource becomes unavailable.

To implement those policies, we consider Rego, a high-level declarative language built for expressing complex policies. Once we have the policy specification and the implementation, we transform both into conditional metagraphs. For this, we develop three generic policy translators: from specification (raw) to specification metagraph, from specification to implementation (Rego), and from implementation-to-implementation metagraph. Those tools enable us to go from a base specification to the implementation of a policy, and then compare them by turning them both into metagraphs.

Policy specification into a conditional metagraph—as denoted ② in Figure 9

To transform this process into a conditional metagraph, we need to define the variables set, the propositions set and the edges set defining the conditional metagraph. To this end, we parse the YAWL file defining the process. A YAWL file is an XML file, from which we can extract relevant information, such as the name of tasks, their inputs and outputs, predicates used for flow control, etc.

The union of elements in the *inputs* and *outputs* of each task make up the *variables set* of the conditional metagraph. The union of predicates of each task make up the *propositions set*. We complete the *edges set* of the metagraph by iterating on all the tasks of the process. We summarize the conditions and actions to take in Figure 12. For each task, irrespective of their order in the YAWL process, relevant elements of the YAWL language are identified. The inputs (outputs) of a task correspond to the invertex (outvertex) of the edge, whereas predicates correspond to propositions in the invertex of the edge. The join code indicates the join operation of the currently processed task, and is used to determine the specific actions in each possible case (AND, OR, XOR). The (non-)completion edge refer to the creation of a new edge from the outputs of the previous task to a proposition signifying the current task has (not) been completed, as we explained earlier with disjoint tasks.

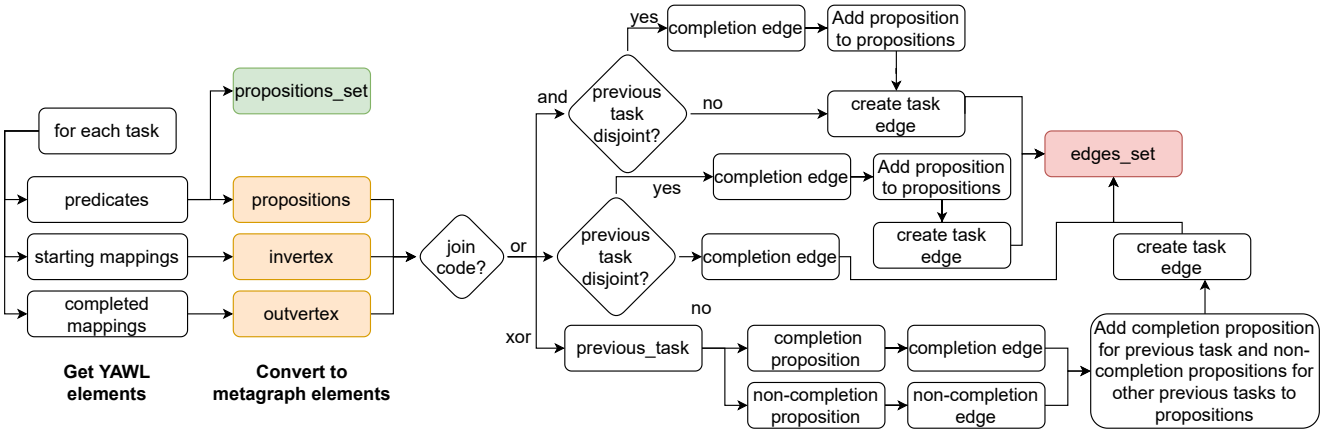


Figure 12. Flowchart of transformations from YAWL to a metagraph. For each task, relevant elements of the YAWL language are identified, and converted to metagraph elements. The join code indicates the join operation of the currently processed task, which is then used to determine the actions to perform in each possible case (AND, OR, XOR).

Figure 11 represents a part of the transformation of our example—the film production process (Figure 10) is translated into a conditional metagraph. Edges represent the tasks of the process, where the invertex (outvertex) represents the inputs (outputs) of the task. As explained in the last section, propositions are added (`_completed` suffix) to make sure that we retain the information of the workflow.

In addition to the YAWL format which specifies workflows, we have also added the possibility of specifying a workflow directly in a metagraph-like format. This form of policy specification can be generically expressed as a list of rules: each describing an edge of the metagraph, as a triplet of the form of $\langle source, destination, policy \rangle$.

To transform this kind of policy specification into a conditional metagraph, we also need to define the variables set, the propositions set and the edge set. To this end, we parse the triplets of the policy specification file. A proposition attached to an edge must be true for the edge to be used in a metapath, thus, an `OR` in a proposition can be viewed as separate edges from the same source to the same destination, with each part of the `OR` becoming a sub-proposition attached to one of the newly created edges. Likewise, the `AND` in a proposition means both parts need to be true for the edge to be used, so the proposition cannot be separated. Those rules will serve as a basis to determine vertices in the metagraph.

In a logical formula, propositions ANDed together are part of the same metagraph edge, whereas propositions ORed together are each part of their own metagraph edge. That is the way conditional metagraphs handle connectivity when considering propositions. A proposition attached to an edge must be true for the edge to be used in a metapath, thus, an OR in a proposition can be viewed as separate edges from the same source to the same destination, with each part of the OR becoming a sub-proposition attached to one of the newly created edges. Considering the proposition between Color and HDR in Figure 8, "$POST\ AND\ (time < 8\ OR\ time > 17)$", we can see the OR that separates $time < 8$ and $time > 17$ is responsible for two different edges when we refine the metagraph representation in Figure 13. Likewise, the AND in a proposition means both parts need to be true for the edge to be used, so the proposition cannot be separated. Considering again the proposition between Color and HDR in Figure 8, we can see the AND means that the POST component of the proposition is a part of both split edges when we refine the metagraph representation in Figure 13.

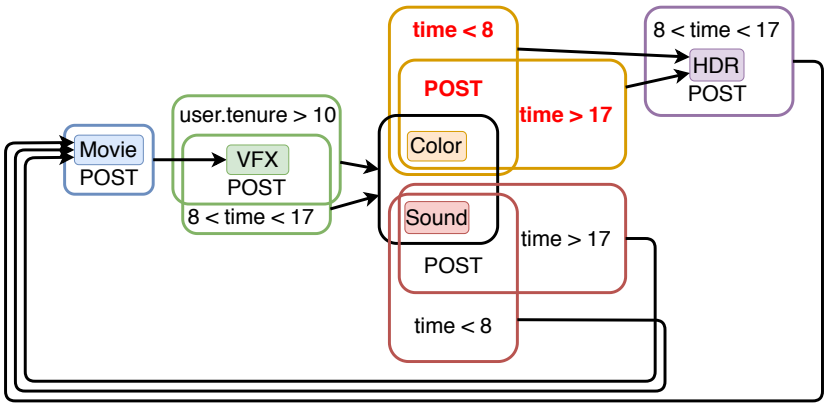


Figure 13. Specification metagraph. Elements of the variable set are identified by filled rectangles, and nodes of the metagraph by unfilled rectangles. Propositions being on the edges or in the invertices are equivalent when dealing with conditional metagraphs.

To fill our proposition set as well as the edge set in our conditional metagraph, we need to turn a given logical formula into its Disjunctive Normal Form (DNF). In DNF, a logical formula is composed of ANDed propositions ORed together, i.e., smaller logical formulas separated by ORs. We can then see that our smaller logical formulas directly correspond to different edges in our metagraph. Take for example the logical formula from Color to HDR in Figure 8: $POST\ AND\ (time < 8\ OR\ time > 17)$. We can transform this expression into its DNF, obtaining $(POST\ AND\ time < 8)\ OR\ (POST\ AND\ time > 17)$. The smaller formulas inside the parentheses correspond to the sub-propositions attached to two edges we obtain in our metagraph (Figure 13).

Then, since each of our smaller logical formulas are either singular atomic propositions or atomic propositions ANDed together, we gather the set of atomic propositions for each edge. Each edge in the conditional metagraph is then generated in the form of a triplet $\langle source, destination, policy \rangle$, where the *policy* component corresponds to one of the smaller logical formulas obtained earlier. We complete the edge set of the metagraph by iterating on all the triplets of the raw specification. The proposition set is simply composed of the union of all atomic propositions, with the generating set being the union of the variable set and the proposition set.

Figure 13 represents the transformation of our example: the movie workflow specification (Figure 8) is translated into a conditional metagraph. Elements of the variable set are identified by filled rectangles, and nodes of the metagraph by unfilled rectangles. Please note that as suggested in the formal definition of conditional metagraphs, we moved the propositions from the edges of the simple workflow graph to the invertices of the

metagraph for a correct and clearer representation on which advanced verification (e.g., checking for conflicts and redundancies) is also possible.

Policy implementation (i.e., Rego) into a conditional metagraph—④ in Figure 9

We use ANTLR4, Another Tool for Language Recognition, which is a parser generator used for translating structured files. After constructing our lexer rules and parser grammar for Rego, we were able to generate the Abstract Syntax Tree (AST) for any Rego policy file. From there, we can walk the AST to generate the implementation metagraph.

Comparing metagraphs—see ⑤

To compare metagraphs, we tag edges in one metagraph upon a match with edges in the other metagraph. Non-tagged edges correspond to errors/mistakes in the implemented policies, singled out by our comparison.

8. Performance Analysis: The Cost of Comparing Random Workflows

To profile our policy verification algorithm, we measure the time required to compare the specification and implementation metagraphs.

8.1. Methodology to Build Random Workflows

We perform such comparisons by varying different elements, namely the number of elements in the workflow, the number of edges in the workflow, the size of the policy on each edge, and the error rate in the implemented policies. Since our verification method compares edges, we measure the computation time as a function of the edges on the metagraph.

To obtain general and representative results when profiling our algorithm, we chose to generate random workflows (instead of relying on few small real cases). This allows us to obtain a general idea of how efficient our algorithm is under various conditions. Since the generation is random for most of the variables defining a metagraph, the generated workflows should not exhibit specific structures or policies that may favor or not the comparison. We thus generate random workflows in YAWL.

In practice, we generate the random workflows by varying these sets of parameters:

- **Size of the workflow**, i.e., number of elements in the generating set: 10, 20, 30, 50 or 100. Those correspond to variables which can be used in the input and output of tasks.
- **Policy size**, i.e., number of conditional propositions on each edge for the policy: 2 or 4.

The number of tasks in the workflow, i.e., the number of edges in the metagraph is a multiple of the number of elements in the generating set, as motivated in the following paragraphs.

Had we used simple graphs, we also would have varied the probability of having edges between any two nodes in the graph (i.e., the graph density), as is customary for Erdős-Rényi random graphs. However, the equivalent density property in metagraphs would be to vary the probability of having edges between any two pairs of possible subsets of the generating set: this leads to a combinatorial explosion of the possible number of edges. This does not sound neither reasonable nor realistic if we compare it with common policy density found in other papers [32,72] or in GitHub projects. Instead, we use the same (static) ratio between the number of nodes and edges that Ranathunga et al. [32] found when they modeled their real-world network policies as a metagraph. They have 1.5 times more edges than the number of elements in the generating set and so do we (to generate our set of workflows). Overall, we generate 30 random conditional metagraphs for each combination of the generation parameters, i.e., the number of elements in the workflow and the policy size, creating 300 different workflow specifications in total (5 generating set sizes, 2 policy sizes, 30 repetitions).

Now that workflow specifications are generated, we need to turn them into their workflow implementation counterparts (i.e., in Rego). As already stated, translating

workflow specifications to their real implementations is error prone. We model this by relying on a given percentage of the elements/propositions in the specification randomly changed to another existing element/proposition. To do so, we consider a last parameter:

- **Error rate**, i.e., fraction of errors in propositions of the metagraph. A value of 0.4 means that 40% of the elements/propositions of the metagraph will be changed to erroneous ones; we consider the following error rates: 0.0, 0.2 and 0.4.

We generate errors randomly, resulting in 30 different Rego files for each workflow specification and for each error rate. We obtain 90 different Rego files per workflow specification in total.

Translation is done by iterating over edges of the conditional metagraph generated from the specification, which will generate the necessary Rego code. Finally, and overall, we obtain 27,000 different policy implementations (300 specifications, 3 error rates, 30 repetitions). The Lines of Code (LoC) of those policy implementations are between 214 and 24,729, which is in line with papers that model real-world policies in terms of LoC size; for example, Ranathunga et al. [32] are citing 5043 for one switch configuration in their case study and researchers in [72] are giving an average LoC size between 125 and 1360.

Now that we have the policy implementations, we can translate those into metagraphs to finally perform the comparison. This is achieved using ANTLR.

Once both specification and implementation metagraphs are generated, we launch our matching algorithm. This algorithm simply compares both metagraphs as two lists of edges. First, for enabling an efficient comparison, we sort both lists by source and destination as respectively the first and second key. Then we try to match edges by iterating through both sorted lists. If both sets are empty after the matching is done, the metagraphs match perfectly. The metagraphs are different otherwise, with the edges remaining in the sets being the ones that are unmatched and the result of errors.

8.2. Evaluation: Sorting Policies and Their Edges has a Limited Cost of $m \cdot log(m)$

To avoid being subject to the bias effects of peak machine load on the CPU time, for each of the 27,000 scenarios, we measure the cumulative time of both sorting and matching for 30 runs. We then end up with a total of 810,000 measures. We ran our measurements on a laptop, equipped with an Intel Core CPU 3.5-GHz, 16GB of RAM and running Ubuntu 18.04.

Among the 30 repetitions, we have a few extreme values. We checked they are due to peak machine load and consequently removed these outliers (i.e., all values with a Z-score superior to three); that leads to remove 9619 values out of 810,000 (1.19%).

Figure 14 represents the time required by our algorithm to detect errors according to the set of parameters in use. As we can see, the error rate has a negligible effect on the duration of the algorithm. On the contrary, as anticipated, the execution time increases with the number of elements in the generating set. The number of elements in the generating set increases the number of edges in the metagraph, by the effect of the 1.5 factor applied on edges. This can be verified in a correlation matrix, which indicates a correlation coefficient of 0.945 between the number of edges and the execution time, and a correlation coefficient of 0.004 between the error rate and the execution time.

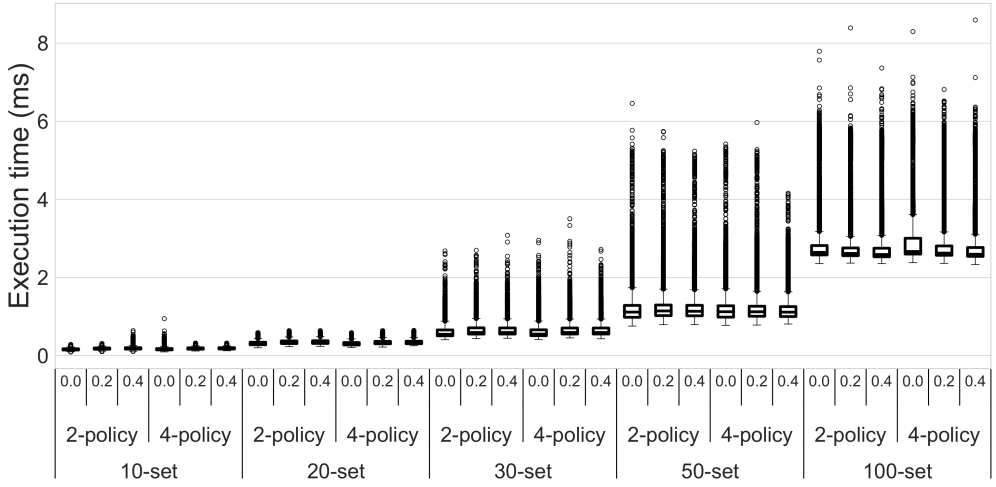


Figure 14. Execution time of our matching algorithm according to several different parameters. The error rate has almost no effect on the execution time, while it increases as expected with the number of elements in the generating set.

The effect of the number of edges on the algorithm time is shown more clearly when looking at Figure 15. It plots the execution time against the number of edges in the metagraphs. The dots represent the mean value for a given number of edges and the red line represents the ordinary least squares regression for the nonlinear function $y = \alpha + \beta \cdot m \cdot log(m)$. We rely on this function as the average time complexity of our sorting (and matching) algorithm is given by $O(m \cdot log(m))$, with m the number of edges. That is the complexity is dominated by the sorting pre-processing (relying on a binary heap for worst cases or using a quicksort-like algorithm for improving average performance). This stage is applied before the actual matching that is then simply linear in the number of edges. When the algorithm time is predicted, we found that the number of edges ($\beta = 0.0020$; $p < 0.001$) is a significant predictor. Indeed, the overall model fit is: $R^2_{adj} = 0.898$, with the *post hoc power analysis* indicating a power greater than 0.999.

In summary, we can argue that our policy verification method can be efficiently implemented as long as the number of propositions in the policy is reasonable. The complete data, code to generate the measures and results, as well as some guidance are publicly available (See https://zenodo.org/record/4912289, accessed on 10 December 2021).

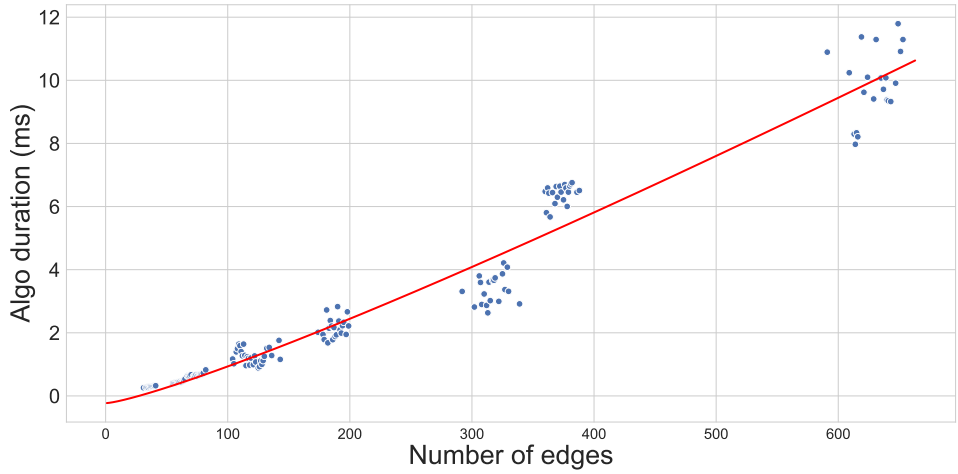


Figure 15. Log regression of the execution time according to the number of edges. When the algorithm time is predicted, we found that the number of edges ($\beta = 0.0020$; $p < 0.001$) is a significant predictor.

9. Conclusions

In this work, we described and implemented a secure architecture that prevents data leaks and protects business intelligence in the cloud. More specifically, in accordance with the principles of zero-trust, we achieve a secure system that enables the exchange of data between non-trusted agents while guaranteeing these data are secure at rest, in transport and cannot be leaked by any agent in both cases. The workflow is defined by the owner and enforced using policy sidecars, which controls the agents participating in the workflow. We realized a proof of concept of our architecture to discuss its benefits and limitations, and monitored key parts of the workflow to show how the data are secured. Our experiments finally show that our approach scales well with increasing workflow complexity.

In addition, we investigated policy modeling and checking as implementations matching their specifications is necessary. We identified metagraphs as an appropriate structure to model those policies, and that it was feasible to transform a YAWL workflow into a metagraph. We detailed to what extent their formal and graphical foundations can guide the reasoning to manipulate such policies and provide means to detect conflicts and mitigate redundancies. This structure is a suitable modeling tool; while it enables policy analysis and allows the search for redundancy and conflicts, we have proposed here to use them for a practical verification of the deployed access control policy regarding its specification. Our proposal compares the initial metagraph specification to its deployed counterpart implementation and reveals their inconsistencies if any. We also evaluate the complexity of our approach to show its scalability.

In the future, we plan to study how changes in the workflow impact the security of the system. Some work could also be done on the removal of trust requirements, by adding Trusted Execution Environments to our infrastructure, or on how to handle frequent user interactions with the system. This would provide us with a fully secure environment, so that even an actor with administration rights on the machine cannot peek at the data. Even though the data in our infrastructure is encrypted at rest, this would also give us another guarantee that the data of the owner and the business intelligence of the contractors are secure for any processing task.

Author Contributions: Conceptualization, L.M., P.M., A.G. and C.P.; methodology, L.M., P.M., A.G. and C.P.; software, L.M.; validation, L.M., P.M., A.G. and C.P.; formal analysis, L.M., P.M., A.G. and C.P.; investigation, L.M., P.M., A.G. and C.P.; resources, L.M., P.M., A.G. and C.P.; data curation, L.M.; writing—original draft preparation, L.M.; writing—review and editing, L.M., P.M., A.G. and C.P.; visualization, L.M., P.M., A.G. and C.P.; supervision, P.M., A.G. and C.P.; project administration, L.M., P.M., A.G. and C.P.; funding acquisition, C.P. All authors have read and agreed to the published version of the manuscript.

Funding: This project has been made possible in part by a grant from the Cisco University Research Program Fund, an advised fund of Silicon Valley Foundation grant number 1318167.

Data Availability Statement: See https://github.com/loicmiller/secure-workflow, accessed on 10 December 2021, for complete data, code as well as guidance to realize our Proof of Concept, test framework, as well as full data, code and statistical analysis in the form of jupyter notebooks. See https://zenodo.org/record/4912289, accessed on 10 December 2021, for complete data, code to generate the measures and results, as well as some guidance on policy verification.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Galov, N. Cloud Adoption Statistics for 2021. 2021. Available online: https://hostingtribunal.com/blog/cloud-adoption-statistics/ (accessed on 10 December 2021).
2. Byers, S.; Cranor, L.; Korman, D.; McDaniel, P.; Cronin, E. Analysis of security vulnerabilities in the movie production and distribution process. In Proceedings of the 3rd ACM Workshop on Digital Rights Management, Washington, DC, USA, 27 October 2003; ACM: New York, NY, USA, 2003; pp. 1–12.
3. Clearinghouse, P.R. Chronology of Data Breaches. 2021. Available online: https://privacyrights.org/data-breaches (accessed on 10 December 2021).

4. Miller, L.; Mérindol, P.; Gallais, A.; Pelsser, C. Towards Secure and Leak-Free Workflows Using Microservice Isolation. In Proceedings of the 2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR), Paris, France, 7–10 June 2021; pp. 1–5.
5. Miller, L.; Mérindol, P.; Gallais, A.; Pelsser, C. Verification of Cloud Security Policies. In Proceedings of the 2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR), Paris, France, 7–10 June 2021; pp. 1–5.
6. Security, R.B. Data Breach Quickview Report 2019 Q3 Trends. 2019. Available online: https://library.cyentia.com/report/report_003207.html (accessed on 10 December 2021).
7. Stempel, J.; Finkle, J. Yahoo Says All Three Billion Accounts Hacked in 2013 Data Theft. 2017. Available online: https://www.reuters.com/article/us-yahoo-cyber/yahoo-says-all-three-billion-accounts-hacked-in-2013-data-theft-idUSKCN1C82O1 (accessed on 10 December 2021).
8. Seals, T. Thousands of MikroTik Routers Hijacked for Eavesdropping. 2018. Available online: https://threatpost.com/thousands-of-mikrotik-routers-hijacked-for-eavesdropping/137165/ (accessed on 10 December 2021).
9. KrebsonSecurity. First American Financial Corp. Leaked Hundreds of Millions of Title Insurance Records. 2019. Available online: https://krebsonsecurity.com/2019/05/first-american-financial-corp-leaked-hundreds-of-millions-of-title-insurance-records (accessed on 10 December 2021).
10. Lecher, C. Google Reportedly Fires Staffer in Media Leak Crackdown. 2019. Available online: https://www.theverge.com/2019/11/12/20962028/google-staff-firing-media-leak-suspension-employee-termination (accessed on 10 December 2021).
11. Jin, C.; Srivastava, A.; Zhang, Z.L. Understanding security group usage in a public iaas cloud. In Proceedings of the IEEE INFOCOM 2016—The 35th Annual IEEE International Conference on Computer Communications, San Francisco, CA, USA, 10–14 April 2016; pp. 1–9.
12. Chandramouli, R.; Butcher, Z. *Building Secure Microservices-Based Applications Using Service-Mesh Architecture*; Technical Report; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2020.
13. Gilman, E.; Barth, D. *Zero Trust Networks*; O'Reilly Media, Incorporated: Newton, MA, USA, 2017.
14. Blog, N.T. Netflix Conductor: A Microservices Orchestrator. 2016. Available online: https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40 (accessed on 10 December 2021).
15. Blog, N.T. Evolution of Netflix Conductor: v2.0 and beyond. 2019. Available online: https://netflixtechblog.com/evolution-of-netflix-conductor-16600be36bca (accessed on 10 December 2021).
16. Valenza, F.; Basile, C.; Canavese, D.; Lioy, A. Classification and analysis of communication protection policy anomalies. *IEEE/ACM Trans. Netw.* **2017**, *25*, 2601–2614. [CrossRef]
17. Moffett, J.D.; Sloman, M.S. Policy hierarchies for distributed systems management. *IEEE J. Sel. Areas Commun.* **1993**, *11*, 1404–1414. [CrossRef]
18. Enterprise, V. Data Breach Investigations Report. 2020. Available online: https://www.verizon.com/business/resources/reports/2020/2020-data-breach-investigations-report.pdf (accessed on 10 December 2021).
19. Amazon. AWS Policy Generator. 2020. Available online: https://awspolicygen.s3.amazonaws.com/policygen.html (accessed on 10 December 2021).
20. Dohndorf, O.; Kruger, J.; Krumm, H.; Fiehe, C.; Litvina, A.; Luck, I.; Stewing, F.J. Tool-supported refinement of high-level requirements and constraints into low-level policies. In Proceedings of the 2011 IEEE International Symposium on Policies for Distributed Systems and Networks, Pisa, Italy, 6–8 June 2011; pp. 97–104.
21. Klinbua, K.; Vatanawood, W. Translating tosca into docker-compose yaml file using antlr. In Proceedings of the 2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, 24–26 November 2017; pp. 145–148.
22. vulners. Razer US: Database Credentials Leak. 2017. Available online: https://vulners.com/hackerone/H1:293470 (accessed on 10 December 2021).
23. Cimpanu, C. Steam Bug Could Have Given You Access to All the CD Keys of Any Game. 2018. Available online: https://www.zdnet.com/article/steam-bug-could-have-given-you-access-to-all-the-cd-keys-of-any-game/ (accessed on 10 December 2021).
24. Muthiyah, L. Hacking Facebook Pages. 2018. Available online: https://thezerohack.com/hacking-facebook-pages (accessed on 10 December 2021).
25. Aboul-Ela, A. Delete Credit Cards from any Twitter Account. 2014. Available online: https://hackerone.com/reports/27404 (accessed on 10 December 2021).
26. Miller, L.; Mérindol, P.; Gallais, A.; Pelsser, C. Towards Secure and Leak-Free Workflows Using Microservice Isolation. *arXiv Prepr.* **2020**, arXiv:2012.06300.
27. Ter Hofstede, A.H.; Van der Aalst, W.M.; Adams, M.; Russell, N. *Modern Business Process Automation: YAWL and Its Support Environment*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2009.
28. Model, B.P. *Notation (bpmn) Version 2.0*; OMG Specification, Object Management Group: Milford, MA, USA, 2011; pp. 22–31.
29. Foundation, Y. YAWL4Film. 2010. Available online: http://yawlfoundation.org/pages/casestudies/yawl4film.html (accessed on 10 December 2021).
30. Blockchain, V. The Future of Business: Multi-Party Business Networks. 2020. Available online: https://octo.vmware.com/the-future-of-business/ (accessed on 10 December 2021).
31. Van Der Aalst, W.M.; Ter Hofstede, A.H. YAWL: Yet another workflow language. *Inf. Syst.* **2005**, *30*, 245–275. [CrossRef]

32. Ranathunga, D.; Roughan, M.; Nguyen, H. Verifiable Policy-Defined Networking using Metagraphs. *IEEE Trans. Dependable Secur. Comput.* **2020**. [CrossRef]
33. Basu, A.; Blanning, R.W. *Metagraphs and Their Applications*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2007; Volume 15.
34. Chandramouli, R. *Security Strategies for Microservices-Based Application Systems*; Technical Report; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2019.
35. El Malki, A.; Zdun, U. Guiding Architectural Decision Making on Service Mesh Based Microservice Architectures. In *European Conference on Software Architecture*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 3–19.
36. Souppaya, M.; Morello, J.; Scarfone, K. *Application Container Security Guide (2nd Draft)*; Technical Report; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2017.
37. Chandramouli, R.; Chandramouli, R. *Security Assurance Requirements for Linux Application Container Deployments*; US Department of Commerce, National Institute of Standards and Technology: Gaithersburg, MD, USA, 2017.
38. de Weever, C.; Andreou, M. *Zero Trust Network Security Model in Containerized Environments*; University of Amsterdam: Amsterdam, The Netherlands, 2020.
39. Hussain, F.; Li, W.; Noye, B.; Sharieh, S.; Ferworn, A. Intelligent Service Mesh Framework for API Security and Management. In Proceedings of the 2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), Vancouver, BC, Canada, 17–19 October 2019; pp. 735–742.
40. Zaheer, Z.; Chang, H.; Mukherjee, S.; Van der Merwe, J. eZTrust: Network-Independent Zero-Trust Perimeterization for Microservices. In Proceedings of the 2019 ACM Symposium on SDN Research, San Jose, CA, USA, 3–4 April 2019; pp. 49–61.
41. Accorsi, R.; Wonnemann, C. Strong non-leak guarantees for workflow models. In Proceedings of the 2011 ACM Symposium on Applied Computing, TaiChung, Taiwan, 21–24 March 2011; pp. 308–314.
42. Shu, X.; Yao, D.D. Data leak detection as a service. In *International Conference on Security and Privacy in Communication Systems*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 222–240.
43. Farhatullah, M. ALP: An authentication and leak prediction model for Cloud Computing privacy. In Proceedings of the 2013 3rd IEEE International Advance Computing Conference (IACC), Ghaziabad, India, 22–23 February 2013; pp. 48–51.
44. Shu, X.; Yao, D.; Bertino, E. Privacy-preserving detection of sensitive data exposure. *IEEE Trans. Inf. Forensics Secur.* **2015**, *10*, 1092–1103. [CrossRef]
45. Liu, F.; Shu, X.; Yao, D.; Butt, A.R. Privacy-preserving scanning of big content for sensitive data exposure with MapReduce. In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, San Antonio, TX, USA, 2–4 March 2015; pp. 195–206.
46. Shu, X.; Zhang, J.; Yao, D.D.; Feng, W.C. Fast detection of transformed data leaks. *IEEE Trans. Inf. Forensics Secur.* **2015**, *11*, 528–542.
47. Shu, X.; Zhang, J.; Yao, D.; Feng, W.C. Rapid screening of transformed data leaks with efficient algorithms and parallel computing. In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, San Antonio, TX, USA, 2–4 March 2015; pp. 147–149.
48. LeVasseur, T.; Richard, P. Data Leak Protection System and Processing Methods Thereof. US Patent 9,754,217, 5 September 2017.
49. Segarra, C.; Delgado-Gonzalo, R.; Lemay, M.; Aublin, P.L.; Pietzuch, P.; Schiavoni, V. Using trusted execution environments for secure stream processing of medical data. In *IFIP International Conference on Distributed Applications and Interoperable Systems*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 91–107.
50. Zuo, C.; Lin, Z.; Zhang, Y. Why does your data leak? Uncovering the data leakage in cloud from mobile apps. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 1296–1310.
51. Jayaraman, K.; Ganesh, V.; Tripunitara, M.; Rinard, M.; Chapin, S. Automatic error finding in access-control policies. In Proceedings of the 18th ACM Conference on Computer and Communications Security, Chicago, IL, USA, 17–21 October 2011; pp. 163–174.
52. Khurat, A.; Suntisrivaraporn, B.; Gollmann, D. Privacy policies verification in composite services using OWL. *Comput. Secur.* **2017**, *67*, 122–141. [CrossRef]
53. Hu, H.; Ahn, G.J.; Kulkarni, K. Discovery and resolution of anomalies in web access control policies. *IEEE Trans. Dependable Secur. Comput.* **2013**, *10*, 341–354. [CrossRef]
54. Koch, M.; Mancini, L.V.; Parisi-Presicce, F. Conflict detection and resolution in access control policy specifications. In *International Conference on Foundations of Software Science and Computation Structures*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 223–238.
55. Schneider, F.B. Enforceable security policies. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2000**, *3*, 30–50. [CrossRef]
56. Cheminod, M.; Durante, L.; Valenza, F.; Valenzano, A. Toward attribute-based access control policy in industrial networked systems. In Proceedings of the 2018 14th IEEE International Workshop on Factory Communication Systems (WFCS), Imperia, Italy, 13–15 June 2018; pp. 1–9.
57. Basile, C.; Canavese, D.; Pitscheider, C.; Lioy, A.; Valenza, F. Assessing network authorization policies via reachability analysis. *Comput. Electr. Eng.* **2017**, *64*, 110–131. [CrossRef]
58. Rezvani, M.; Rajaratnam, D.; Ignjatovic, A.; Pagnucco, M.; Jha, S. Analyzing XACML policies using answer set programming. *Int. J. Inf. Secur.* **2019**, *18*, 465–479. [CrossRef]

59. Attia, H.B.; Kahloul, L.; Benhazrallah, S.; Bourekkache, S. Using Hierarchical Timed Coloured Petri Nets in the formal study of TRBAC security policies. *Int. J. Inf. Secur.* **2020**, *19*, 163–187. [CrossRef]
60. Liu, A.X.; Chen, F.; Hwang, J.; Xie, T. Xengine: A fast and scalable XACML policy evaluation engine. *ACM Sigmetrics Perform. Eval. Rev.* **2008**, *36*, 265–276. [CrossRef]
61. Liu, A.X.; Chen, F.; Hwang, J.; Xie, T. Designing fast and scalable XACML policy evaluation engines. *IEEE Trans. Comput.* **2010**, *60*, 1802–1817. [CrossRef]
62. Hughes, G.; Bultan, T. Automated verification of access control policies using a SAT solver. *Int. J. Softw. Tools Technol. Transf.* **2008**, *10*, 503–520. [CrossRef]
63. Bera, P.; Ghosh, S.K.; Dasgupta, P. Policy based security analysis in enterprise networks: A formal approach. *IEEE Trans. Netw. Serv. Manag.* **2010**, *7*, 231–243. [CrossRef]
64. Ranathunga, D.; Nguyen, H.; Roughan, M. MGtoolkit: A python package for implementing metagraphs. *SoftwareX* **2017**, *6*, 91–93. [CrossRef]
65. Hamza, A.; Ranathunga, D.; Gharakheili, H.H.; Roughan, M.; Sivaraman, V. Clear as MUD: Generating, validating and applying IoT behavioral profiles. In Proceedings of the 2018 Workshop on IoT Security and Privacy, Budapest, Hungary, 20 August 2018; pp. 8–14.
66. Hamza, A.; Ranathunga, D.; Gharakheili, H.H.; Benson, T.A.; Roughan, M.; Sivaraman, V. Verifying and monitoring iots network behavior using mud profiles. *IEEE Trans. Dependable Secur. Comput.* **2020**. [CrossRef]
67. Docker. Docker. 2019. Available online: https://www.docker.com/ (accessed on 10 December 2021).
68. Kubernetes. Kubernetes. 2020. Available online: https://kubernetes.io/ (accessed on 10 December 2021).
69. Istio. Istio. 2020. Available online: https://istio.io/ (accessed on 10 December 2021).
70. Envoy. Envoy. 2020. Available online: https://www.envoyproxy.io/ (accessed on 10 December 2021).
71. Open Policy Agent. Open Policy Agent. 2020. Available online: https://www.openpolicyagent.org/ (accessed on 10 December 2021).
72. Ranathunga, D.; Roughan, M.; Nguyen, H.; Kernick, P.; Falkner, N. Case studies of scada firewall configurations and the implications for best practices. *IEEE Trans. Netw. Serv. Manag.* **2016**, *13*, 871–884. [CrossRef]