UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Configuring edge device provenance through messaging middleware

A CamFlow and MQTT system implementation

Tarald Eide Øines

Master thesis in INF-3990 15th May 2024

Supervisors

Main supervisor:	Dagenbord, Håvard J.	UiT The Arctic University of Norway, Faculty of Science and Technology, Department of Computer Science

Abstract

Integrity of data is important in today as more of societies structure today are distributed and becomes vulnerable to dishonest entities on their edge devices. Using provenance to prove integrity over edge devices and distributed networks is difficult, as it often produces big amounts of data which fills up storage without having a need to be used. Comm2Prov seeks to fix this by combining the selective provenance capture of CamFlow and the distributed messaging of MQTT. Using a client to send messages between underlying CamFlow and the MQTT service, Comm2Prov allows commands to be sent to CamFlow instances running on edge devices. This can be used to issue tracking control, allowing distant auditors to turn off and on provenance on devices without having to need direct access to the device itself. Through this thesis, the challenges and problems which were discovered during Comm2Prov's creation will be discussed, including topics as: security, performance, and the issues with provenance.

Contents

List of Figures

List of Tables

1

Introduction

Unprotected digital data is vulnerable to manipulation by non-trusted entities Integrity of data is an underlying issue which becomes more rampant as more data is put into the digital world. Data can be generated in tremendous volumes at tremendous speeds. As such, it can become difficult to decide whether the data generated is genuine data or non-genuine data generated by a variety of reasons.

With massive data generation, genuine data normally outnumbers non-genuine data making the overall average give a realistic representation than individual data. However, this data assumption does not work when individual data points are more important than the overall. For an analysis of data, the overall average is often more importance than individual pieces of data, making occasional non-genuine data entries of less importance. But for an entity which depends on individual data pieces, the non-genuine data can be disruptive or harmful for their purpose.

Several methods have been developed to keep the integrity of data, with their own advantages and disadvantages to them. Unchangeable distributed ledger methods such as Blockchain is an example which can keep the integrity of data put on the chain. But this technology meets issues as the chain grows longer.

Data provenance is the records of digital data's ownership and changes over time and can be used as a method to increase the integrity of data. It is

derived from the idea of provenance, where ownership is documented and authenticated. Provenance defines as the history of an object and it's change in ownership, originally used for keeping record of ownership transfer between valued art or work. Data provenance is considered the history of digital objects, records of its transformations and derivations over time. Depending on the provenance standard, data provenance may also record agents entities associated with activities, linking them to user(s) or group(s) responsible for transformations or derivations.

Using the records or provenance, post-event inspection can be performed to see the history of data. Recording these events allows on to later inspect and validate the origin of some data. This can be used for a variety of goals, such as identify and detect unwanted behaviour within a system or inspect the authenticity of data. Provenance itself is not a preventative measure, but a post-analytic measure to discover possible faults within the system, allowing countermeasure to be put in place with more detailed hindsight knowledge. In such ways has provenance has been used in several systems for various purposes. Rapsheets [1], P-Gaussian [2], and Kairos [3] are examples on systems using provenance data for intrusion detection and persistent threats. FRAPpuccino [4] is an example of using provenance for fault detection during runtime. Provenance has even parts in systems like Cobweb [5] who are used to create remote attestation.

But data provenance has the issues of having high storage and overhead on a system to gain complete image over the transformations of data. Depending on the activity on a audited unit, they can create large volumes of data in short time. This makes provenance auditing less attractive to use by default, as more data means that analysis will take longer and longer.

For provenance systems where capturing a full image of the entire system is desired to produce an effect, such as detection systems where monitoring the device is the goal, it is difficult to find compromise on provenance data generation. Rapsheets' [1] converts traditional provenance graphs into trimmed threat alert graphs to save up on long-term log retention for this reason. Meanwhile, systems like Cobweb [5] which does not use provenance for detection can compromise limiting what it captures in echange for the full picture.

1.1 Problem Statement

Edge devices provide entrance to larger networks or systems which are closed from outside environments. These entry points are necessary for the system to interact with the outside world, giving them data which the systems needs

to operate. But it is here with these entry points that the integrity of data become harder to keep intact. As all data created outside of the system is near impossible to verify. Moreover, if these entry points are infected or compromised in some manner, the data these entry points may introduce can be falsified or manipulated. One cannot also rule out the possibility of malicious activity of independent entity within the system might affect the data provided.

Distributed systems which have edge devices as entry points spread over large distances can be hard to monitor whether or not the devices act the way that they should. Malicious actors who have gained access to the network may impersonate edge devices through methods such as spoofing, making spotting them hard within the network. Moreover, with several different edge devices it might be hard to pinpoint which device false data comes from.

Provenance is defined as the history of an object and its change in ownership, commonly used for keeping record of ownership transfer between valued art or work. Data provenance is considered the history of digital objects, records of its transformations and derivations over time. Data provenance can also include records of the agents associated with activities, linking them to user(s) or group(s) responsible for transformations or derivations.

Using the records or provenance, post-event inspection can be performed to see the history of data. This can be used for a variety of goals, such as identify and detect unwanted behaviour within a system or inspect the authenticity of data. Provenance is not a preventative measure, but a post-analytic measure to discover possible faults, allowing countermeasure to be put in place with more detailed hindsight knowledge.

Provenance auditing can be used to improve the integrity of singular data, as it can record the transformation of data as it enter the wider network through edge devices. However, constant auditing of edge devices is less desirable for this purpose, as the auditing will cause unnecessary overhead on non-compromised devices. Furthermore, with several edge devices adding their own data to the system, the audits will unnecessarily create provenance data for auditing. This increases the storage overhead and time needed for inspecting the audit logs later. Constantly auditing all edge devices when a only a sub-set of data is wanted over some period of time is inefficient for the network and post-analysis. But configuration of provenance capture can be difficult as auditors may not have direct access to all edge devices.

Our thesis is that you can improve integrity of data through provenance in a network without having constant overhead on audited devices. Our system, Comm2Prov, is designed on the principle of configuring edge device provenance capture through messaging with the assumption that you do not have direct

access to the edge devices. In Comm2Prov, auditing does not need to be activate by default, reducing overhead which can be noticeable on smaller edge devices. Provenance Auditing is activated on suspected edge devices by an auditor after the suspicion has been established, as provenance is tool more favoured for analysing unwanted behaviour in systems post-happening than to discovers it.

This thesis will look into the challenges of provenance collection across different edge devices through testing and analysis of provenance capture in different configurations. The results and challenges encountered will be discussed, as will potential further solutions and considerations of them.

2

Background

The previous chapter was a short introductions to the concepts, problems, and solutions this thesis will provide. This chapter will see more in-depth background on the concepts this thesis builds itself upon.

2.1 W3C-PROV

The W3C-PROV standard [6] provides a general model for classifying provenance data, alongside optional specifications and further classifications. The W3C-PROV Standard is centered around three key concepts: entities, activities, agents: Entities range from conceptual things to physical entities that are recorded in the provenance records. Activities are defined as actions that bring entities into existence, modify existing entities, or make use of entities. An agent is an existence ranging from an organization of people to simple software or other entities who has a role in an activity, giving them some level of responsibility for that activity. Agents can be seen as an attribute within provenance, which can be applied to activities and entities to show "who" bears responsibility for them. Roles are description of how an "entity" played part in an activity, specifying the relationship between them.

PROV does not specify roles, as roles are to be specific for their instances which may vary. When an entity changes partly, they become a derivation of their previous instance. In PROV, each result of a derivation is considered a

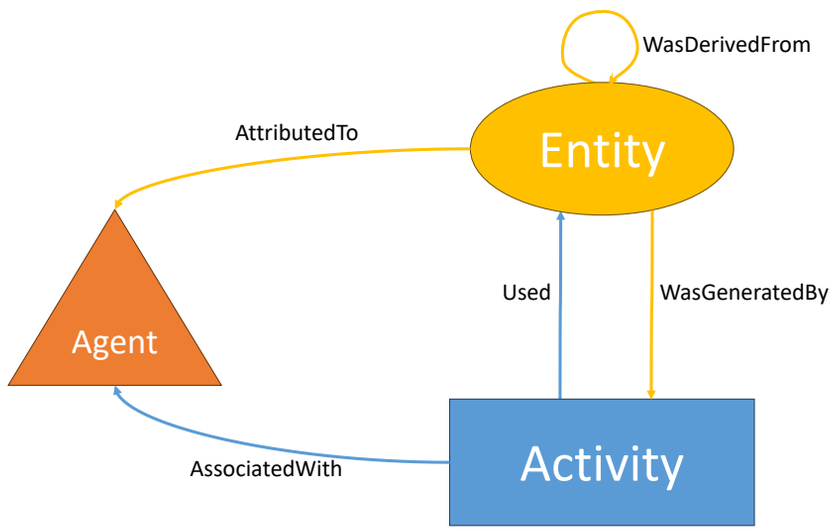

Figure 2.1: Cyclic graph example of W3C-Prov.

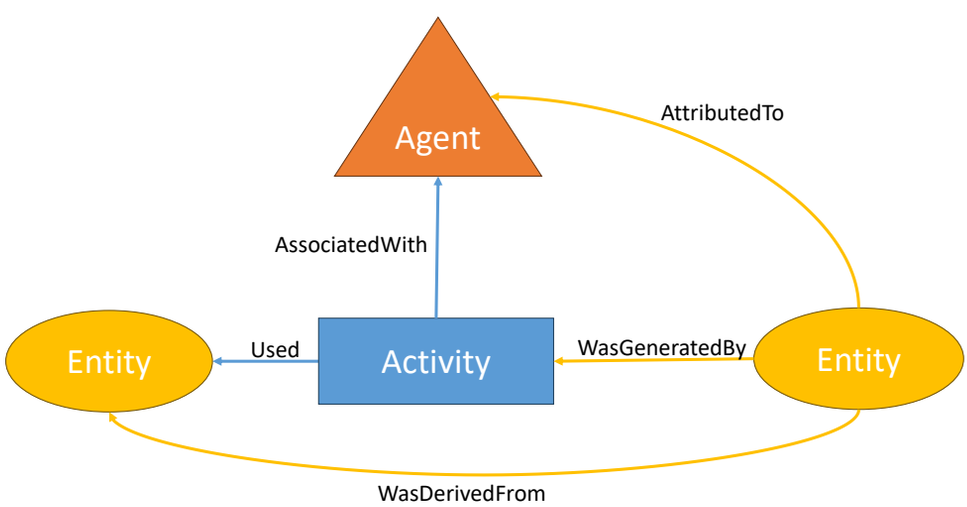

Figure 2.2: Acyclic graph example of W3C-Prov.

new entity, and some common and specialized derivations can be specifically described such as revision and quotations. Plans are pre-determined and pre-defined procedures that an agent can follow in execution of activity. Time is often a crucial part of provenance and a staple description within W3C-PROV model, allowing timing of start and finish of events within the provenance records. W3C-PROV allows for specialization of entities, allowing alternate versions of an entity to be recorded as a specilization of another entity.

W3C has more specifications and optional additional model specifications to follow.It displays provenance mostly be treating entities as input to an activity or another entity, with the output result being a derivation. Agents are assigned through Linux User and Group delegation system, allowing agents to take up roles and responsibility for entities and activities.

2.2 CamFlow

CamFlow[7]is a kernel-level provenance capture mechanism using Linux Security Module and Netfilter hooks to capture provenance. It was developed for the purpose to be easily integrated into Platform as a Service cloud module. CamFlow is entirely open source[1] and available for public.

CamFlow has been called state-of-the-art within provenance capture by previous papers [8][9] citing and using the CamFlow system. It has been used in several systems detecting Advanced persistent threats using provenance [10][11] as a component. It has also been used in as a component in other types of systems [12][13][4].The main benefits of using CamFlow its high configurability and open-source availability, allowing it to be tailored and used just the way a system would want it.

The decision to use CamFlow also comes from its high configurability, allowing specific provenance capture in order to reduce the amount of provenance data collected. This is one of the core concepts this thesis seeks to explore.

2.2.1 Development

CamFlow was originally proposed by Thomas F. J.-M. Pasquier, Jatinder Singh, David Eyers, Jean Bacon in 2015 as an Information Flow Control system. The paper being republished in 2017[14] . CamFlow set out to do this using metadata to control data flow, implemented as a Linux Security Module (LSM) in the OS

1. https://github.com/CamFlow/

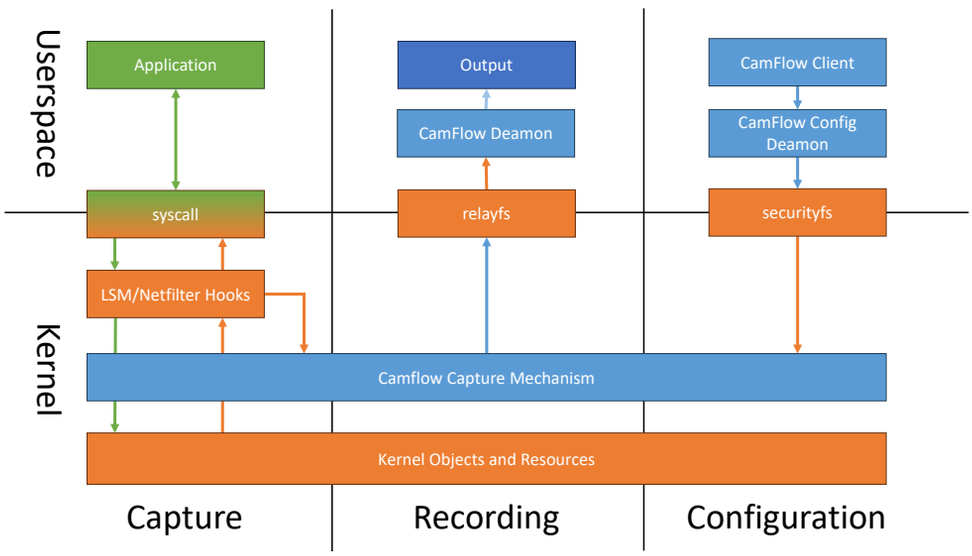

Figure 2.3: Architecture and flows of CamFlow.

with IFC-enabled middleware compatibility. Metadata methods implemented to constrict data flows in CamFlow are labeling and taint tagging with checkpoints to avoid data leakage. In the follow up paper for CamFlow [15] released in 2016, they continued on with the IFC development, but started to discuss Information Flow Auditing (IFA) as a byproduct of IFC. IFA became relevant for CamFlow as it was demonstrated in the MrLazy paper[16] that provenance records could be used to verify IFC constraints in a system.

This prompted the later shift from Information Flow Control to Information Flow Audit in CamFlow, as published in the latest CamFlow paper [7] which released in 2017, the diverged from IFC to IFA by converting CamFlow to a auditing system. Building upon the previous established hooks for the IFC system, CamFlow converted to log the provenance data instead. The strategy for CamFlow is that it should be easy to maintain, use existing kernel mechanism whenever available without duplication, and manageable integration into mainline Linux. This strategy came from the developers earlier experience with trying to maintain PASS[17] and LPM[18].

2.2.2 Architecture

There are four components which make up CamFlow: the CamFlow client, the CamFlow recording deamon, the CamFlow configuration daemon, and the CamFlow capture mechanism integrated into the OS kernel.

The CamFlow capture mechanism in kernel uses the OS reference monitor which captures kernel calls from applications. CamFlow relies on Linux Security Model (LSM) hooks and Netfilter hooks to capture kernel calls without the applications being aware of it. Two types of hooks are used, a hook to catch when kernel objected is allocated and a hook used to catch when a kernel object is accessed. CamFlow capture over sockets mostly contain metadata of the packets caught, but may additionally be configured to capture the packet payload. These captures are published to the Relayfs [19] pseudo-file system, which will be retrieved by camflowd.

CamFlow daemon is responsible for converting the capture to provenance log recording in the specified format to the specified output. Currently, CamFlow daemon supports two different provenance formats: the W3C-PROV-JSON format and the SPADE JSON format. The SPADE JSON format is a provenance data model based on W3C-PROV, but created for the SPADE [20] system. The 5 possible output configurations for the CamFlow daemon are: null, mqtt, unix_socket, fifo, log. Null discards the log records. Mqtt publishes the records to a MQTT broker in which CamFlow connects through on startup.Unix_socket publishes the records to a UNIX socket. Fifo publishes the records to a fifo pipeline. Log stores the records locally on a log file in text format.

Camflow configuration deamon is responsible for configuring the capture of the CamFlow system. The daemon loads its configurations from a init configuration file when the system starts up, putting it into the kernel through securityfs interface. Camflow configuration deamon can also reconfigure some capture configurations through the CamFlow client. Camflow client can be interacted with through the commands entered thorough a command line, with the correct authority. allowing dynamic capture configuration during runtime.

2.2.3 Design

One of the main features of CamFlow is its high configurability, allowing the enabling and disabling of provenance capture of nearly all hook events. This allows for enabling provenance on only sensitive data or data of interest, avoiding filling the logs with uninteresting provenance data. This makes the provenance records smaller and easier to digest by human and machine, at the cost of the overall picture that comes with full provenance. In contrast, one can also disable provenance on hook events by making them opaque. CamFlow uses this opaque function on its own components to ensure that it doesn't recursively captures a capture. Additionaly, CamFlow can enable security context[2] using the main Linux Security Module format.

2. https://selinuxproject.org/page/NB_SC

CamFlow can configure whether to capture provenance or not based on hooks for: Filters set on nodes and edges, specific programs and directories and files, specific processes, and specific network activity. Provenance can also be enabled for Linux users and groups within the system, tracking their associated activities as agents in the system. The hooks trigger upon system calls, where an application wishes to access a kernel object. This kernel object hook-based capture makes CamFlow unable to capture in-application events, confining the capture granularity to be process-level at the finest. In addition to hooks provided by LSM and Netfilter, CamFlow has implemented some of its own hooks[3] for the purpose of better provenance. Their full details on supported node/vertices hooks[4] and relation hooks[5] can be found on their github.

CamFlow has to major capture settings in which it can monitor a system: whole-system provenance and partial-system provenance. Whole-system provenance captures all hooks called to kernel, fully tracking all non-opaque entity kernel calls on the OS. Partial-system provenance which allows one to configure the level of provenance by enabling provenance on the hooks. Provenance data might become to big and clotted if all events are recorded. In contrast, limiting the recording might miss events of interest, losing the complete image of a systems data flow.

The provenancce model CamFlow uses is W3C-PROV, and SPADE-PROV [20] which is based on W3C-PROV. CamFlow extends the W3C-PROV model for provenance data with it's own additional attributes, used for a more complete provenance image in CamFlow system. An example of this is the *machine_id* attribute, being an unique identification in provenance logs for the machine which the events occurred on. This ID is based on the linux hostid[6] if not manually configured.

As stated on their website [7], CamFlow can enable different levels of capture once activated. CamFlow can use standard tracking on an entity, capturing calls done in relation to the entity that is tracked. It can also propagate track an entity, where it tracks data flows coming out of the entity. Data flows coming in are limited to their entrance into the tracked entity, as it cannot back-track the flow from before it was set to be tracked. The propagation tracking can be halted through the data flow entering a propagation filtered node or relation. Additionally, IPv4 tracking can be set to record a message's package content on specified incoming and outgoing IPs and ports.

3. https://github.com/CamFlow/camflow-dev/blob/master/docs/HOOKS.md
4. https://github.com/CamFlow/camflow-dev/blob/master/docs/VERTICES.md
5. https://github.com/CamFlow/camflow-dev/blob/master/docs/RELATIONS.md
6. https://man.linuxexplore.com/htmlman1/hostid.1.html
7. https://camflow.org/

Entities in a system can have labels and taint applied to them. The labelling systems originates from the earlier days of CamFlow, where they were used to enforce Information Flow Control. Labels are sets of tags applied to entities, which propagate to activities and other entities as the labeled entity interacts with them. The tags are applied to data as additional metadata In previous versions of CamFlow, the labels were split into integrity and secrecy, restraining information from flowing between higher security security context and lower security contexts. Tags and privileges to create them are stored as 64-bit opaque fields within the kernel object, with only active entities have mutable tags and privileges.

Taint follows a similar model as labels, where a tag taint is spread and propagated through every entity it passes through, but only checked at certain "sink" points in order to prevent leakage. The "sanitizing" process of taint data can be processes like encryption of data or stripping of sensitive information in the data, depending on what is deemed necessary. The use of CamFlow's IFC against CamFlow's taint system is question of performance or quality.

CamFlow cannot track flows happening within processes, especially processes which have a shared state with others. Without intra-application tracking of the memory usage, it is unsure whether data from the shared state is spread or not. Therefore, incoming and outgoing flows of a process with a shared state is assumed be incoming and outgoing flow from that shared state.

2.3 MQTT

MQTT is an Pub-Sub Messaging standard protocol created by Oasis [21] for IoT device messaging over the standard TCP/IP protocol. Pub-Sub messaging systems uses a subscription model to transfer messages. Subscribers subscribes to a topic, which a publisher may publish a message too. The subscriber receives the message published by the publisher on their subscribed topics through an intermediary. The intermediary might be other entities in the network, but is commonly an independent broker which handles the topics, subscriptions, and messages.

MQTT is designed to by lightweight, making it ideal for edge devices which can have low bandwidth and computing power. There are different version of the MQTT protocol, the original being introduced in 1999. MQTT v3.1.1 is the most commonly used version today, with MQTT v5 currently seeing limited use in applications.

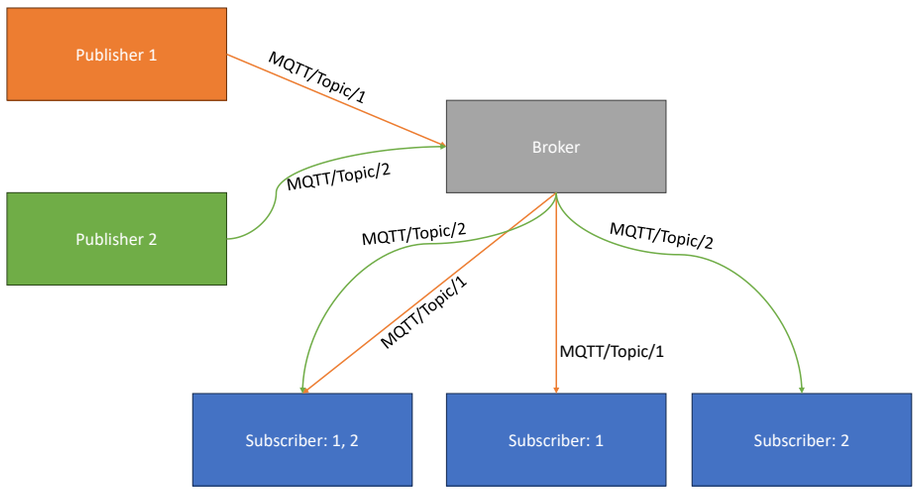

Figure 2.4: Architecture of MQTT

2.3.1 Architechture

MQTT uses a messaging broker to handle messages, topics, and subscribers to those topics. MQTT messaging consists of three different entities in the network it sends messages over. A broker which works as an intermediary for publisher and subscribers. The subscriber who subscribes to a topic through a broker, and receives all published messages on the topic on the broker. The publisher publishes a message on a topic to a broker, which then gets further published by the broker to all subscribers. Publisher are not aware of who subscribes to the topics, and subscribers are not aware of who publishes to their topics. Broker stores who subscribes to which topics until connection is broken, unless client session is not set to be cleaned after disconnect.

2.3.2 Security measures

MQTT supports authentication a security measures along with other security measures, which broker configure and clients comply to.This varies a bit with broker implementation, but without it the broker has no control over who subscribes to a topic and who publishes to a topic. The subscriber has no knowledge or control who publishes to it's subscribed topics. Likewise, the publisher has no control or knowledge of who subscribes to the topic they publish.

Payload can be encrypted at application level, without any connection with

the broker. Messages to the brokers can also be TLS or SSL protocol protected using self-provided certificates, free commercial certificates or apid certification.

All clients provide some sort of client ID upon connecting with a broker, allowing the broker to restrict access based on provided client ID. MQTT brokers may be configured to require a simple login system upon connection, where a username and password is required. The handling of this login information might vary from broker implementations, being from simple plaintext transfer to encrypted login. It may be possible to deploy certificate authentication by using x509 client certificates. But it requires handling of several certificates by the brokers.

Topic subscription and publishing can be restricted. This restriction can be tied to User or Client ID, but not x509 certificates as of now. User login can be used for restricting access to topics, allowing only certain topics for certain users. This access restriction can be applied to three level, with general restriction, Client ID restrictions and User restrictions.

2.3.3 Topic

Messages sent over the MQTT broker is split into topics in order to divide messages into groups. Topics can have sub-categories to further divide them into smaller groups by using "/" as a level divider, similar to url divisions for sub-domains. There are few restrictions on topics, such as topic must be UTF-8 strings, case sensitive, and contain minimum a single character to be valid. The only standard topic is the "$SYS" topics, which is used for standardised commands allowing clients to ask the broker for its state. Topics are created as subscriber subscribe to them or publisher publish to them. If there are no active subscribers or publisher on a topic, the topic is discarded.

There are two signs reserved for wildcard use, plus symbol ("+") and hashtag symbol ("#"), which allows one to subscribe to multiple topics within the wildcard. Plus symbol is a single level wildcard, only usable for variations of the level it is employed on. Hashtag is a multi-level wildcard, subscribing to all variants on the level it is applied and all sub-level variants. Plus sign can go in between levels in a topic, while hashtag can only be used at the end of a topic. Wildcards are not allowed to be used for publishing, forcing specific topic publication.

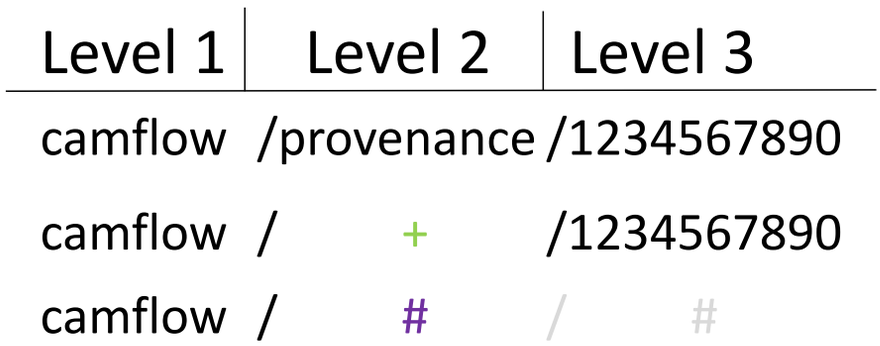

Figure 2.5: Example of MQTT topic construction with wildcards.

2.3.4 Message structure

Messages published contain at minimum a topic, Quality of service value and a payload which is the message itself. Quality of service tells which delivery protocol is to be used for the message. The options for message QoS are maximum once, minimum once, and exactly once, deciding how certain the publisher is taht the message has been delivered to the broker. With maximum once, the message is only sent once to the broker with no follow up. With minimum once, the message is sent expecting an acknowledgement that it has been delivered, repeating the message if one isn't returned. With exactly once, an acknowledgement that no more duplicate messages are to be sent from the sender is sent, while the receiver responds acknowledging it.

Clients may also leave a Last Will Message, which is a message that is automatically published once the client disconnects. The message is sent to the broker and stored upon client connecting, being published by the broker once the client disconnects for any arbitrary reason.

A message can be flagged to be retained, meaning that it will be kept by the broker and given to new subscribers of the topic. Only one message can be retained at a time for a topic, pushing out the old one when a new one is published. Retained messages remain even as new messages are published as long as their messages are not flagged to be retained. Messages to have been retained are automatically sent to new subscribers of the topic upon their subscriptions.

Clients are default set to have clean sessions with the broker, where the broker does not keep non-retained messages for subscribers who have not received the published message. But the clients may set their session to be unclean sessions, where the broker will keep messages until for the clients subscribed topics if the clients where to be disconnected. The result of delivering the message may also vary depending on the QoS setting of the message.

3

Comm2Prov

This thesis introduces Comm2Prov, an system which uses messaging middleware to connect provenance from instances separated over varying distances. Comm2Prov seeks to to challenge a widespread provenance approach, providing an example off controlling provenance through messaging for potential edge units. Comm2prov bases itself upon CamFlow provenance capture and MQTT messaging protocol.

3.1 Design

3.1.1 Architecture

Comm2Prov is designed to be used for devices connecting to a another part within the network with a Comm2Prov client, providing information towards a central database of a kind. Combining the usage of MQTT for it's low-cost messaging of the selective provenance of CamFlow, the goal is to create low-cost provenance auditing network. With the focus on low impact on the device, the system could seeks to be suitable for edge devices with regular to low computation power.

There are three types of devices within the Comm2Prov architecture: Provenance devices, MQTT broker, and the auditor. An example can be seen in the Architecture figure 3.1. Devices with provenance are machines which runs Cam-

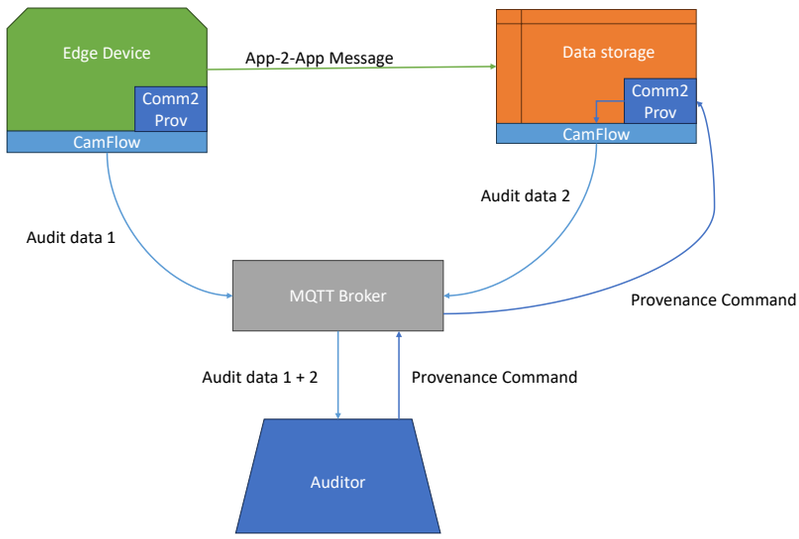

Figure 3.1: Architecture of Comm2Prov.

Flow and the pub-sub client in provenance mode. Auditors are agents which runs client in command mode, where they can give out provenance commands and receive provenance records from provenance devices. The auditor receiving provenance records and the auditor giving out commands do not need to be the same device, as any auditor may give out commands and subscriptions as they are permitted. Comm2Prov uses standard MQTT brokers without any specific modification to their operating.

CamFlow remains an underlying component of the system Comm2Prov is implemented, where only the pub-sub client needs to be aware of its existence. The pub-sub system does not provide the records to the broker as of the current implementation, as CamFlow is able to provide the data itself. This allows the pub-sub system to avoid being declared as opaque in the CamFlow system to avoid reporting loops, allowing the pub-sub clients actions to be recorded.

The Comm2Prov architecture is not limited to single broker, as is portrayed in the figure 3.2. Multiple brokers may be used to spread workload and responsibility across several unit. With this spread in responsibility and workload, some brokers may be designated with higher security constraints than others.

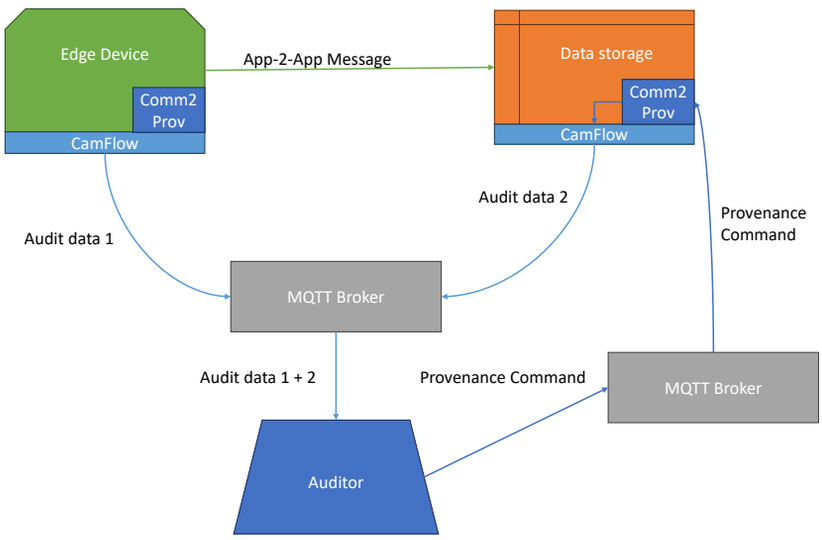

Figure 3.2: Possible network setup of Comm2Prov.

Level 1	Level 2	Level 3
camflow	/provenance	/{machine_id}
camflow	/status	/{machine_id}
camflow	/order	/{machine_id}

Figure 3.3: Topic structure used in Comm2Prov.

3.1.2 Messaging

MQTT uses topics for which devices subscribe and publish to. This topic creation is dynamic, appearing as they are used by subscribers and publishers, and discarded as they become unused.

Comm2Prov expands upon the topic used for CamFlow provenance, "*camflow/provenance/{machine_id}*", with two new categories on the second topic level. As demonstrated in the figure above3.3, these two new categories beside "*provenance*" are "*status*" and "*order*". The topics structure used inside a Comm2Prov system network are static and identical, except for the machine-id used to identify provenance delivering clients. The topic structures may be different than what is used here, as the CamFlow MQTT topic can be configured. But the machine-id is put on the last level in the provenance reporting topic, as it is implemented as such in CamFlow and cannot be configured.

Category "*provenance*" is used by surveyed devices to deliver their provenance records to subscribed auditors or storage. It is the standard topic used by CamFlow to directly send their provenance data over MQTT.

The "*status*" category is used to maintain a retained message with the current status of the device. Currently, there are only two relevant statuses used: Running and Stopped. Theses statuses provide knowledge whether the device is online or not. The message is retained so new connecting auditors receive the status upon subscription. The status list can be expanded to include more information about the device, but those can depend on what is needed from the system. Comm2Prov keeps it simple for demonstration principles.

The final category, the "*order*" category, is used for the edge device client to listen for orders to activate or deactivate provenance. Through this, provenance can be activated without direct connection to devices. If the edge device where to be configured to subscribe to their order topic with unclean sessions, they will receive commands sent while they were offline upon re-connection.

3.1.3 Comm2Prov Client

The pub-sub client is two distinct components of the systems. Provenance publisher client (pub) which work as a daemon application waiting for orders to activate or deactivate order from the MQTT broker it is connected to. Provenance subscriber client (sub) is the auditor client which initiates provenance on pub clients by sending specific order across the MQTT service. Both clients are essentially independent of one another, only sharing the same MQTT messaging syntax for topics and commands on the MQTT device. This means that

different versions of each client can interact and be used on the same network without issues while they use the same messaging syntax.

The pub client acts similarly to a daemon, initiated and connected to a predefined MQTT broker at the start of the system it is installed on. With the connection, the pub client sets up it's own topics using it's own CamFlow generated machine-id. Using the topic structure mentioned in Comm2Prov Topic figure3.3, the pub client subscribes to its own order topic and publishes their current status to their status topic. The provenance topic is reserved for provenance deliver by CamFlow. CamFlow should automatically connect to the MQTT broker on their provenance topic upon started.

Pub client uses its own CamFlow order topic and posting to it's status topic. When it receives an order, it will execute the appropriate command through a command line call. The CamFlow client needs to be called in this way, as CamFlow API does not provide more CamFlow interaction than application level provenance supplications to CamFlow.

In order for the provenance client to be receive orders, it needs to subscribe to a topic using it's own CamFlow generated *machine-id*. However, CamFlow does not provide any means in which to extract a dynamically generated CamFlow *machine-id*. Machine ID' may be preset per machine, allowing static implementation of Machine IDs into the device, which require specific tailoring per device. Comm2Prov's extracts a machine's local CamFlow generated machine ID through extracting it from a CamFlow provenance generated log. CamFlow generates an initial provenance log entry upon device startup, being the boot of the machine. However, this requires CamFlow to generate a local provenance log in addition to directly publishing to MQTT server.

For auditing devices, an intractable command line interface is provided in the as the client. The auditing client can perform actions such as subscribing to device provenance, requesting available devices, and sending orders to activate provenance on devices.

In order for client to subscribe to a topic, it needs to know available provenance devices. MQTT has no option to know which topics are live on the broker, unless you modify the broker. In order to for a client to know all available topics on the broker, it needs to subscribe to all available topics. Since our system designates a topic to device status, *camflow/status/+* is used to extract all available systems with their information. Active provenance clients have their status message declared as active with a topic containing their CamFlow Machine ID. In cases a provenance client goes offline, their last will message to their status topic will relay their inactivity.

3.2 Implementation

3.2.1 Requirements

There are three components needed to for the system: the Comm2Prov client, CamFlow, and a MQTT broker.

CamFlow was built to be run on fedora [1] operating system, with the possibilities of running on other linux based systems without any guarantees. Thus the minimun requirement is for Comm2Prov to be deployed on Linux based operating systems, with Fedora operating system being the optimal deployment.

Dependencies

The Comm2Prov client was implemented in Rust [2] and Paho-MQTT **??** [3]to run. The implementation uses the additional Rust crates of Regex [4]and lazy_static [5]. Although Rust lazy_static and Regex are technically optional, they simplify some parts of the Comm2Prov client.

The provenance and auditor client was tested on the virtual machine fedora OS, and was not tested on other opterating systems. It is expects some compatibility with linux-based OS systems, but there is no guarantee for other OS. The provenance client requires to be on a linux based OS, as its basis is CamFlow. The auditor client does not require a specific OS to run, only the needed dependencies. in actuality, the auditor does not need to be our implemented client, as long as it follows the rules in which Comm2Prov operates

Assumptions

TheComm2Prov client is assumed to not unexpectedly during the operating system is active. If it were to crash, then it will be restarted when this is discovered. If the entity were to restart, the pub-sub client will automatically restart along with it. The Comm2Prov client is also assumed to be started as a starting process during or shortly after startup.

Camflow assumes that it is installed on trustworthy devices. For simplicity, this assumption will carry on in the Comm2Prov system, as Comm2Prov does not

1. https://fedoraproject.org/
2. https://www.rust-lang.org/
3. https://docs.rs/paho-mqtt/latest/paho_mqtt/
4. https://docs.rs/regex/latest/regex/
5. https://docs.rs/lazy_static/latest/lazy_static/

provide any addition environmental security.

3.2.2 Provenance

CamFlow is used as the provenance capturing system, as it allows for run-time configuration of provenance capture. The reasoning for using CamFlow is because it has high configurability and support some, something the proposed system Comm2Prov build upon. A bonus is that CamFlow provides direct MQTT publishing of its provenance data, avoiding storage of provenance data locally on installed devices. The version of CamFlow used is the CamFlow version 0.9.0 and libprovenance v0.5.5, as these are the most recent version during the work of this thesis. Comm2Prov start out with the initial provenance configuration of CamFlow, letting any further configuration be set as a case requires it.

3.2.3 Comm2Prov Client

Client to use and send commands through MQTT (v3.1.1) was implemented in rust using the Paho-MQTT library implemented for rust. Considering how Comm2Prov would be considerable to use on more constrained edge hardware, C and Rust were the contending implementation languages. Rust came on top due to it's memory management being more user-friendly and thus more secure to work with.

Compiling the pub-sub client provide two different executable files, "sub" and "pub". Pub executable is the Comm2Prov provenance client which waits in the background for orders published to the MQTT broker it is connected to. Sub executable is the auditor client, representing a client which sends commands and receive provenance to the MQTT broker which a Comm2Prov provenance client has attached to.

Pub Client

The Pub executable is the provenance client which listens for orders and activates provenance. When the Pub client is initiated, it gathers its own CamFlow *machine_id* to subscribe and publish to the correct topics. The *machine_id* is either statically saved as a pre-defined setting for the Pub client, needing synchronous tailoring between the specific CamFlow settings and Pub client.

Otherwise, it can be extracted from the a command line call or a log entry. This requires CamFlow to activated to store logs in a readable text format in addition to sending the audit data over MQTT. A log entry is reads an entry

the audit log created by CamFlow, always containing at minimum a entry upon startup. The log reading do require an assumption or knowledge of how the logs are stored within the system.

Using the command line extract takes advantage of CamFlow command options *-s* to extract the CamFlow provenance settings, including the *machine_id*. It does this by running *camflow -s* a command line using std::process:Command library at necessary privilege level, saving the output in a string.

In either options, Regex is used twice to find and extract the specific *machine_id* number in the string containing the entry or command output. First Regex extracts the *machine_id* entry with the ID, while the second Regex extracts the *machine_id* ID only. The *machine_id* does not change unless the system is restarted with new CamFlow settings, the CamFlow *machine_id* is stored as a non-volatile variable to be used throughout the run-time of the Pub client.

Afterwards, it configures a MQTT client using MQTT-Paho *CreateOptionsBuilder* to create and optimize the client as wanted. The server url and MQTT client-id the is pre-defined in the client configuration settings, set up before compilation. It connects using a synchronous client, assuring it finishes setup in specified order. After the client is optimized, it is finalized and created to serve as the Pub client.

With a client created, the Pub Client attempts to connect to a broker by building a connection using MQTT-Paho in a independently running thread. Most options, such as the address of broker to connect to, is pre-defined by the client configurations and set up before compilation. The broker the Pub client can be the same as the broker that CamFlow provides direct provenance to, but does not need to be. Sessions are set to unclean so that the Pub Client can reconnect to the same broker, resuming the session. This can be used to receive orders which were sent while disconnected once the Pub client has reconnected. A last will message is set to be sent to the broker upon sudden disconnection, used for signaling to any new connectors that the Pub Client is offline. The last will message is a retained message to the Pub Client's own "*camflow/status/*{*machine_id*}" topic with the payload for signaling that the Client is offline.

When the connection options are finalized, the Client connects to the broker using the options described options. Firstly it publishes the status that it is running to its "*camflow/status/*{*machine_id*}", signaling that the Pub Client is up. Secondly, it subscribes to order topic "*camflow/order/*{*machine_id*}" using the *machine_id* extracted from CamFlow. After subscribing, it immediately starts waiting for messages sent over the order topic until the Pub client is shut

down. At this point the Pub Client will act like a daemon until it receives a message across the topics.

When it receives a message over "*camflow/order/{machine_id}*" topic, the message payload is extracted and run through a dictionary to match command with orders. This is to avoid direct connection with The dictionary is a static hash with a mutex lock on it, used to convert orders received into provenance altering actions. A specific command in string format is used to hash out a string containing the command for changing capture policy. So far, only the capture policy of fully enabling or disabling CamFlow whole tracking is implemented. More commands can be implemented, but such commands would require tailoring or a more complex message delivery and reception with a structure to separate command hash and arguments. Once it has successfully passed through the dictionary and gotten the command, it is executed by calling a Command Line in-process and executing the extracted dictionary specified command through it. After execution, Pub client goes back to waiting for next message sent over the command topic.

Sub client

The sub executable is the auditor client used for sending orders to Pub clients. Client is intractable through the command line which started the auditor client, using prompts for sending commands.

When initiated, the Auditor client attempts to sets up a MQTT-Paho client which to interact with the MQTT broker through. The MQTT client takes a pre-defined broker address and client id from the settings. After setting up the client, the client tries to open a connection to the specified broker. The connection opens with no last will message and a clean session, as the auditor does not need to be remembered by the broker.

Upon successful connection, the Auditor client waits for input through the command line. There are seven options implemented currently for the Auditor client, which are: "stop", "order", "refresh", "receive", "norecieve", "subscribe", "unsubscribe". The "stop" option shuts down Auditor client, cutting connection with broker and the exiting the process.

"refresh" gets a list of running provenance clients connected to the server. However, MQTT does not have any implemented feature to list who is connected to the broker or which topic is active on the broker. In order to get them, the Sub Client subscribes using to the *camflow/status/#* topic, receiving statuses of all provenance brokers connected. As the statuses are retained messages, they are immediately received upon connecting. Then it goes through every

message, looking for a message matching the status to the one signifying the broker is running. On a match, the *machine_id* is extracted from the topic using Regex and stored in a vector for later. After going through every retained message, the client unsubscribes to the *camflow/status/#* topic. The new vector of living provenance clients replaces any previous vectors of living provenance clients that might be outdated.

"subscribe" is used to subscribe to the provenance of a provenance machine. Upon selecting subscribe option, the *machine_id* of all living provenance clients stored from "refresh" option vector is listed. Inputs are taken through a loop, repeating to store several *machine_id*'s until the input to continue breaks it. One can choose to subscribe to all topics, subscribing to all listed provenance clients. From the list, the *machine_id* of the provenance client to subscribe too can be input one after another and is stored in a list of subscribing topics. If the *machine_id* to subscribe to is already in the list or not listed amongst the living machines, it is discarded. Alternatively, the subscribing process can be halted and will return to the other options. Once the *machine_id*'s to be subscribed to is finished, the QoS of the subscriptions are to be selected. All provenance client subscriptions use the same selected QoS . Once the provenance clients and QoS have been selected, they are subscribed too. The subscribed topics are stored in a vector.

"unsubscribe" option works similar to "subscribe". Instead of selecting subscription topics from living machines, the stored subscribed topics are used to select them. Any topic chosen to unsubscribe from is removed from the subscribed topics vector. One can choose to unsubscribe to all topics, removing all subscribed topics from the vector and unsubscribing to them. Once selected, the topics are unsubscribed.

"receive" starts the reading of messages subscribed to by the client. The reading of messages is put into a thread, allowing further options to be selected. There is only one thread, as all messages published by all topics are read at the same time. Currently, there are no definite resolution to what happens to the messages. The thread will continue running and receiving messages until stopped. "norecieve" option ends the thread by signaling the running thread to stop.

"order" is used to send orders to available provenance machines. First, the machines to send order to are selected. Using the vector of running provenance clients as a list, one can select a client or all clients to send order to. Afterwards, the full command to send to the provenance client(s) is taken as an input in. For each machine selected, a message is created and sent on the topic with the order in it. Checking for correct syntax on the order is done on the provenance clients side, as they decide which orders are available.

3.2.4 MQTT

Comm2Prov has a basic messaging implementation, which does not require any specific MQTT broker to service it. As long as the chosen MQTT broker satisfies the MQTT protocol, it should be viable. However, in the search of potential MQTT broker, Mosquitto [6] was noticed as a potential MQTT broker implementation. This is mostly due to mosquitto being open-source[7], the same as CamFlow, providing potential for further development of Comm2Prov. Mosquitto is also compatible with the Paho-MQTT library for Rust.

6. https://mosquitto.org/
7. https://github.com/eclipse/mosquitto

4

Experiments

4.1 Environment

The virtual machines were run on a Windows 11 Pro OS version 22H2, and a 13th Gen Intel(R) Core(TM) i7-13700 2.10 GHz processor with 128 GB available ram. Simulators used were using virtualbox (V0.7) [1] and vagrant (V2.3.7) [2]. Experiments were run using the vagrant environments set up by CamFlow developers (https://github.com/CamFlow/vagrant). More specifically on the Spade vagrant environment, setup for CamFlow with SPADE V3.0 [20] on a fedora 33 cloud-edition OS. The virtual machine was upgraded from CamFlow from version 0.7 to CamFlow version 0.9, while the latest libprovenance version 0.5.5 installed alongside it.

Each virtual machine has designated 24576 bytes of memory, 4 CPU's with execution cap set to 70 in the vagrantfile settings. Additionally, 80 GB of disk space was configured for each device, although they do not necessarily need that much. 40 GB of disk space should be sufficient, where the 80 GB were initially set due to potential growth of provenance logs.

A vagrant private network was configured between the virtual machines, giving each machine its own IP address on the network. The different virtual machines used this network to communicate which each other, simulating a private

1. https://www.virtualbox.org/
2. https://www.vagrantup.com/

network Comm2Prov could potentially be deployed on. Although for our case the virtual machines were run on a singular physical machine.

4.2 Benchmarking provenance

The purpose of running a benchmark experience was to evaluate CamFlow's system impact and its ability to capture provenance during high computation loads. It also evaluated and compared different provenance capture states CamFlow can enable. Additionally, the accuracy and provenance data generation was recorded to further illustrate how large volumes of provenance data can be generated.

4.2.1 Implementation

CamFlow was tested with Machine Learning Benchmark, specifically using the Penn Machine Learning Benchmark (PMLB)[22] as it was most ease of use on the virtual machines. This test was run to see how CamFlow would be able to record data flow and it's impact on performance during high intensity computing. Another aspect was to evaluate if provenance tracking is viable to use in Machine Learning settings. PMBL is a collection of several datasets, collected as a benchamrk suite pre-proccessed to be more easier to use, as the data itself is less important than the machine learning benchmarking.

The pre-processsing of the dataset and functions leave some of abstractions for the benchmark process, with their impact on possible overhead being less clear. The specific of what PMLB contains can be found on the epistasislab git hosted website [3]. However, with the singular purpose of benchmarking through Machine Learning algorithms, the content of used datasets are less relevant with the benchmark results being the final matter. As for the purpose of our experiment is to put laod on the machine and system, the abstractions are non-consequential for our purpose. Although, these abstractions do make it harder to get a clear image of overhead causes on the finer granularity.

The test uses the example in the PMBL github repository [4] with some minor modifications to it. The machine learning itself was done using the python scikit-learning module, as done in the example. The machine learning methods used from the scikit-learning module were logistic regression and guassian blur. The PMLB datasets were run using the scikit-learning modules, using

3. https://epistasislab.github.io/pmlb/
4. https://github.com/EpistasisLab/pmlb

logistic regression and gaussian blur training models. The number of iterations in which the models were trained varied depending on the experiment. As some datasets in PMLB are significantly large, the machine learning models fails completing some larger of them. But as it continues the process of training on different datasets, this does not impact the experiment, as the training is more important than the results of the training.

Time was measured by noting current time before machine learning initiated and after both training models had completed. The amount of iterations and subsequently the amount of datasets visited are noted as well, to have clearer image of the amount training which was done.

A bash script was used to repeatedly run the experiment, allowing for additional information gathering from the system. From this bash script, the size of provenance data was captured from a log file generated by CamFlow by using the Linux command to read the metadata of the file. The results of the command where stored in a text file for later use. Additionally, the amount of dropped events from CamFlow where extracted similarly be storing the result of the "camflow –drop" command.

As the log file or the dropped events recorded could not be reset during testing, they accumulate as the testing proceeded. The final numbers where calculated by using the difference between the previous extracted record with the current extracted record.

4.2.2 Results

The average results of the experiment is described in benchmark table4.1. There were some instances of unusual results, such as the drops occurring only on one round across each propagate and whole tracking test. This inflates the average of drops from what usually was 0 drops to 1/10 of the singular incident as seen in graph 4.6. Additionally, the propagation seems to have a larger log during the first round than it should, which could be correlated to the drops detected.

Some of the graphs starting values and ending values were adjusted to better display the individual differences.

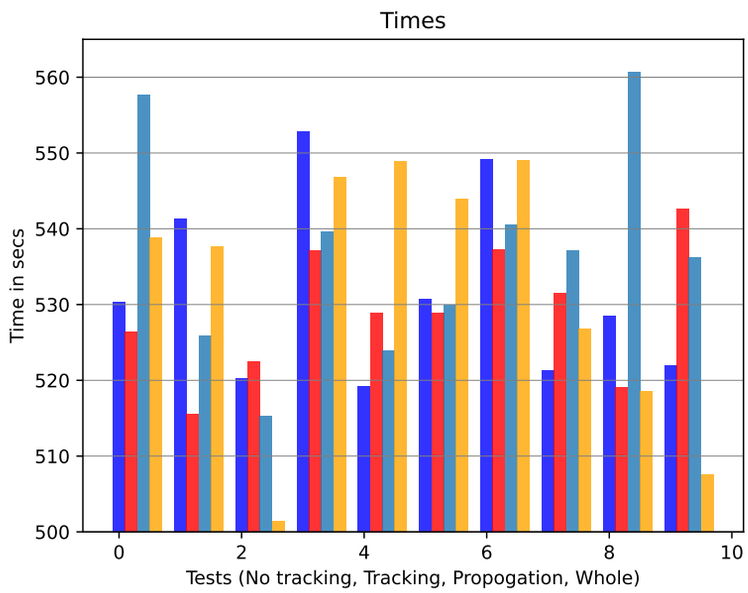

Figure 4.1: Benchmark times comparison figure

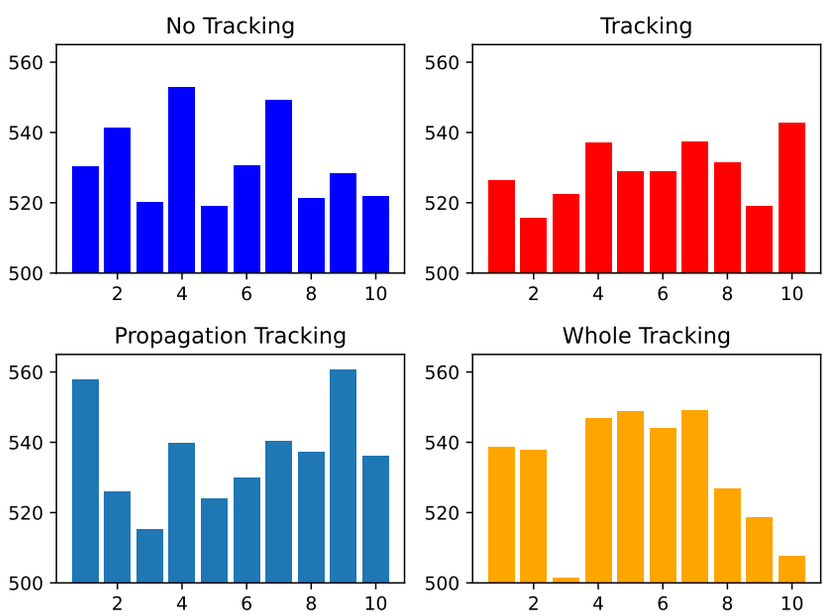

Figure 4.2: Benchmark times individual figure

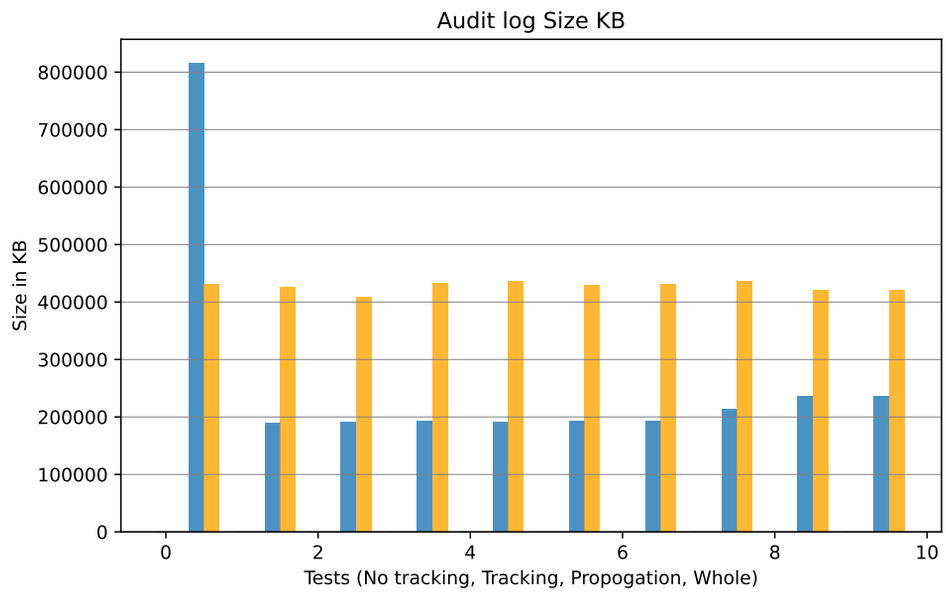

Figure 4.3: Benchmark size comparison figure

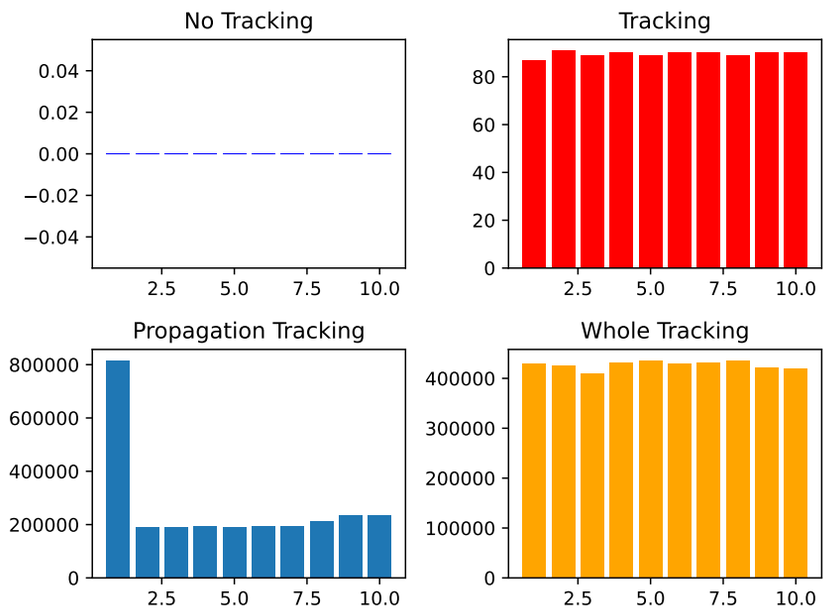

Figure 4.4: Benchmark size individual figure

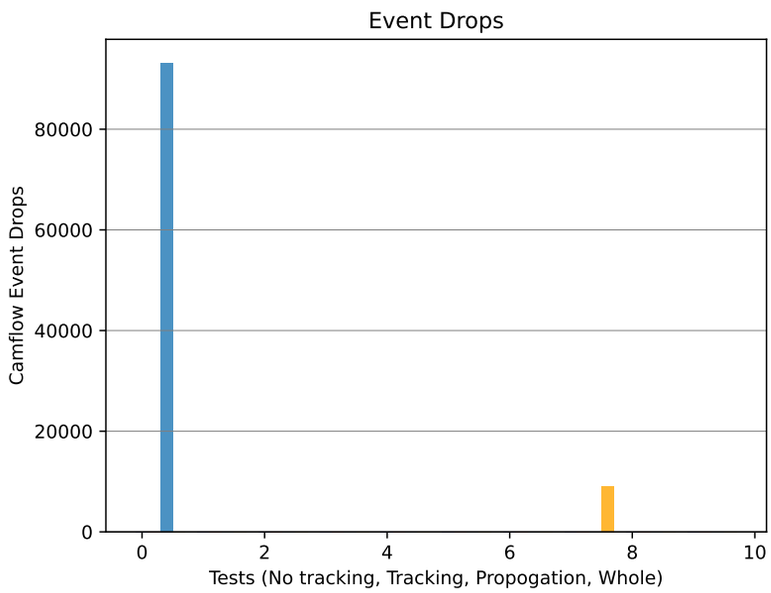

Figure 4.5: Benchmark drops comparison figure

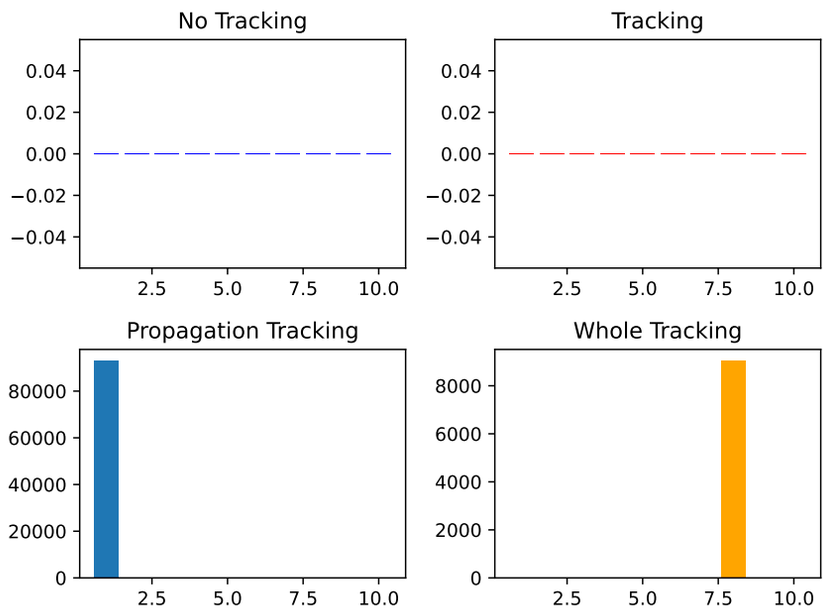

Figure 4.6: Benchmark drops individual figure

Tracking level	AVG. Time	AVG. Size KB	Drops
No tracking	531.5678623	0	0
File	529.0043206	89.5	0
Propagation	536.6962618	265093.4	9319.8
Whole system	531.9642351	427109.9	906

Table 4.1: Average result of ML Benchmarking

4.3 Network transfer tracking

The purpose of this experiment was to test the impact of capturing messages sent over a network. While the previous experiment tested the impact of capture during heavy computation, this experiment tests the capture of network messages.

This experiment was conducted to find the overhead caused by CamFlow provenance capture of message system with a test to send multiple messages over a socket. Similarly to the benchmark experiment, different settings of provenance capture were tested. More notably, CamFlow's ability to capture network packages were tested to see it's impact and accuracy with a constant message flow.

The experiments were done using two vagrant virtual machines, one for sending data and one for receiving data. The machines were setup individually in order for CamFlow to generate different identifications for each machine. Although setup individually, they are setup using identical procedures to keep their likeness. Communication was done over a private network setup by vagrant, each machine specifying their network address.

4.3.1 Implementation

The test was implemented in Python v3.9.9.0 using python sockets to send messages over a private created vagrant network. A sender and a recipient are connected through unix sockets using Python sockets. Connection happens through a static IP configured beforehand put in as values. Time was measured by the experiment implementation by noting down the difference in time before and after the test started. The test was run separately on both machines, one configured to send data to the one configured to receive it.

To server as fodder for the messages, the content of a file was read and sent over the network in which the recipient recorded the contents down in a file. The file sent was a text document containing "Lorem Ipsum" paragraphs repeated

has a file size of 1.0662956238 GB.

Upon starting up, each side attempts to establish connection to each other. Receiving end waits for the sender to connect to them, while the sender immediately tries to connect to the receiver. This interactions warrants the receiving end to initiate before the sender, possible creating overhead as it waits.

Upon gaining connection with the receiver, the sender opens the lorem text file, sending lorem paragraphs from the file line by line. It does this until it reaches a limit set upon starting the test or the end of the of the lorem data file. Messages are sent as plain text over the socket. The receiver reads these files and writes them into a local file. As the messages received are stacked in queue, and the write operation takes longer than the read operations, the receiver occasionally writes multiple lines at ones into the file. The receiver keeps writing messages into the file, stopping once it receives a message with nothing in it. Once sender and receiver are done sending and receiving, they write the time and amount of sent or received messages before disconnecting and exiting.

Scripts were used to successfully run the experiments multiple times in a row. In order for the sender to not attempt to connect earlier than the receiver can wait for them on repeats, an two second wait was added between on the sender side. This created some artificial overhead on the receiver side which has to wait two seconds or less for the sender. Additionally, the script measures the provenance audit logs size and CamFlow dropped provenance packages between each test. The size of the audit log was captured in KB while the amount of dropped packages were extracted using the CamFlow drops command.

Six different CamFlow settings were tested in this experiment. First four were the same as the ones used in the benchmark: not tracking, tracking messaging process, propagate tracking the messaging, and whole system tracking. All of these were tested one after another on with the same script, without resetting the CamFlow capture mechanism and log. As the provenance logs size and amount of CamFlow dropped events are cumulative across the tests, the final result of these is calculated using the difference between previous instance and current instances.

The last two types of test were network message capture tests, with and without package capture. Tracking IP address through CamFlow requires an IP address, mask for the IP address, and a a port number in which to track. CamFlow can distinguish between packages received or sent over the IP connection, as ingress or egress. Both machines were configured to track receiving and sent

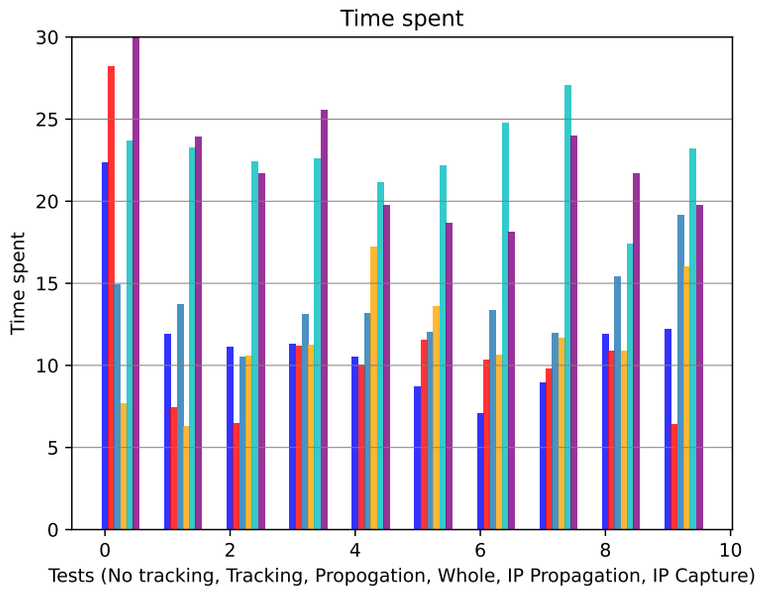

Figure 4.7: Transfer send time comparison figure

messages over the same IP and PORT. Tests were run the same as the previous tests, time captured during the test and the cumulative audit size and drop amount taken through calculation.

As the receiver takes multiple messages at once, final message can be lost in the stack as the receiver reads mulktiple messages at once. To ensure that the ending message is received alone, the ending message is sent after a second delay, creating additionaly overhead. This generates enough time for the receiver side to write the final message. Without this stop, the receiving end might miss the concluding message, looping indefinitely waiting for message the message that was missed.

4.3.2 Result

The average results of experiments are listed in table 4.2. During these experiments too, there were some differing edge case results, which were calculated into the average. The tables are split into the receiver and sender side, showing both sides of the experiments.

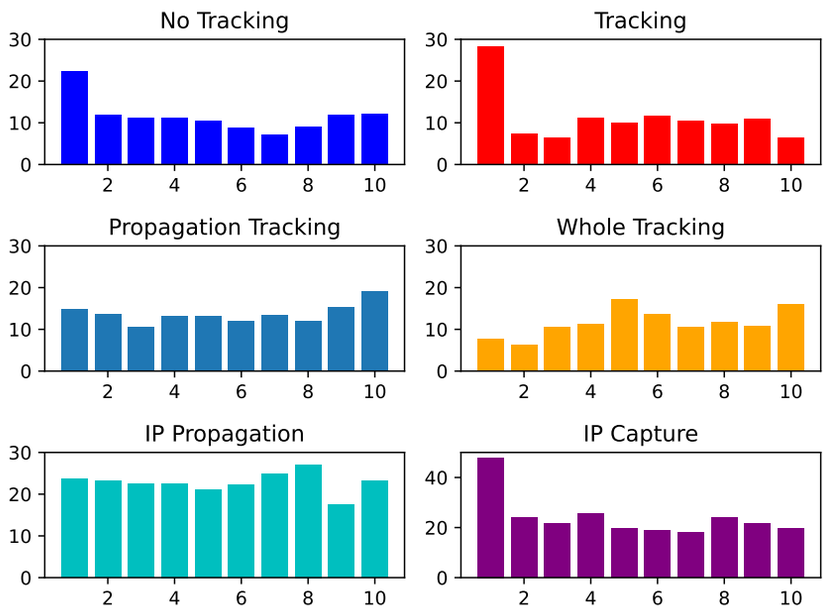

Figure 4.8: Transfer send time individual figure

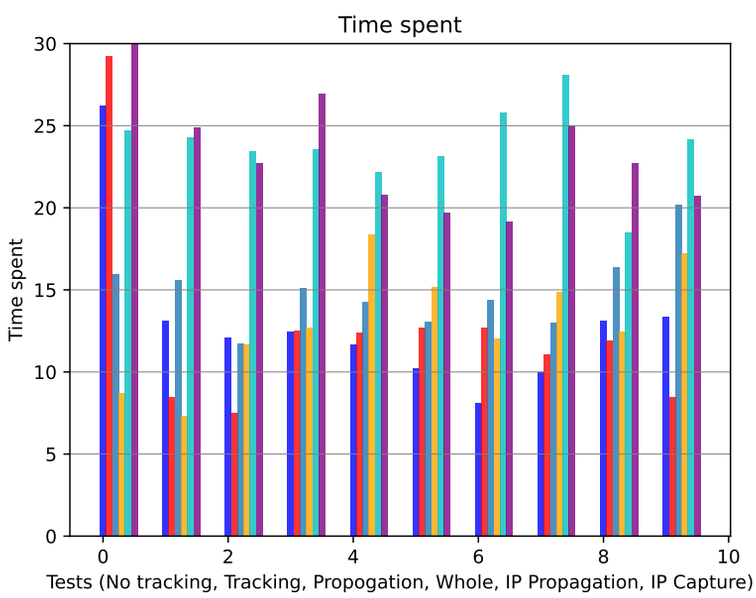

Figure 4.9: Transfer receive time comparison figure

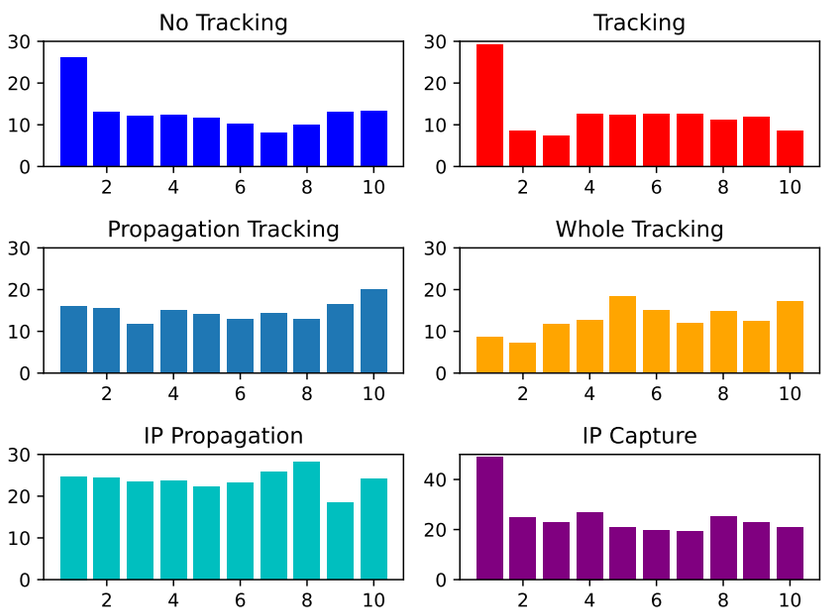

Figure 4.10: Transfer receive time individual figure

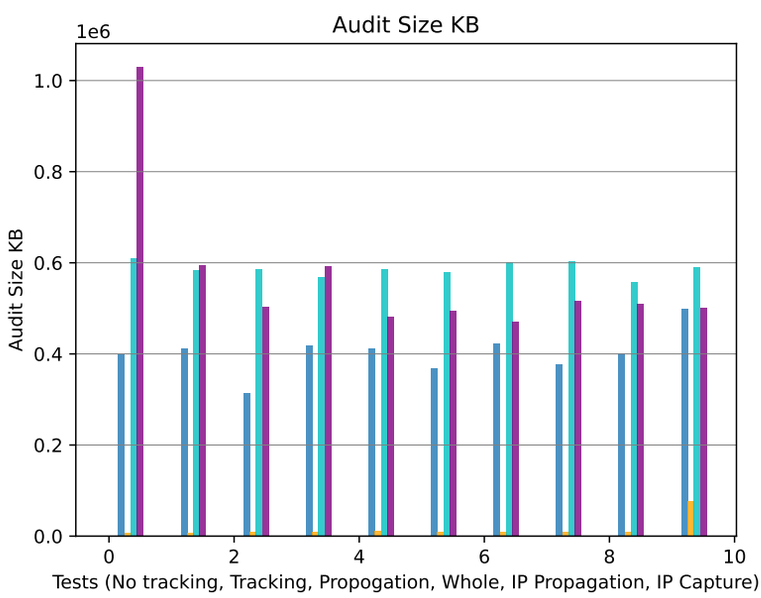

Figure 4.11: Transfer send audit size comparison figure

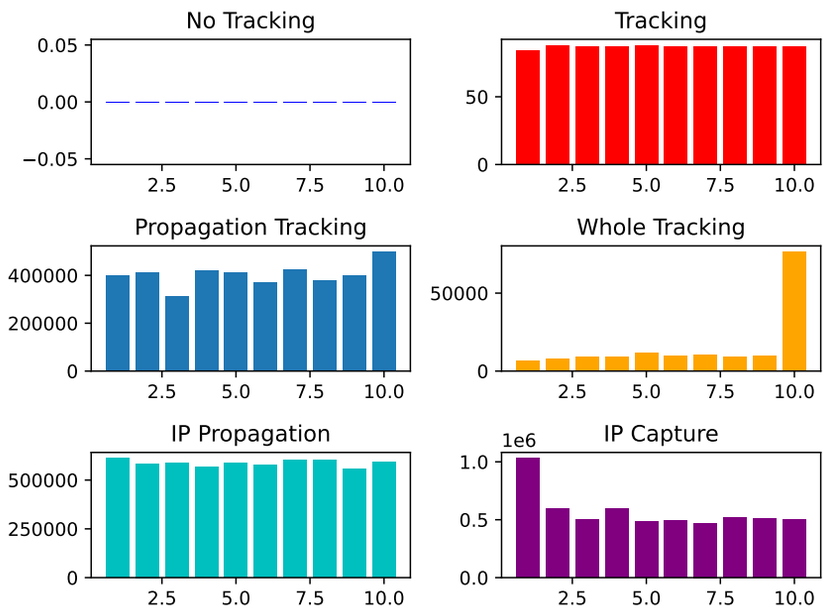

Figure 4.12: Transfer send audit size individual figure

Figure 4.13: Transfer receive audit size comparison figure

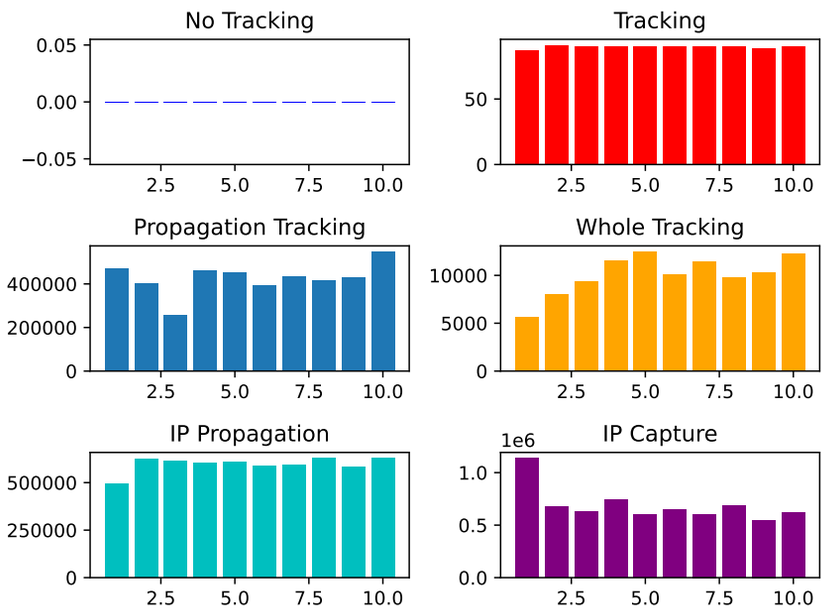

Figure 4.14: Transfer receive audit size individual figure

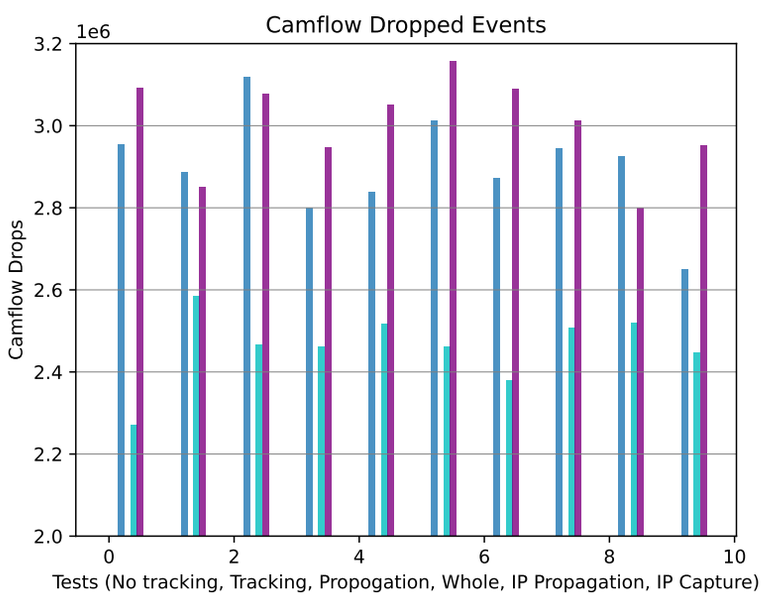

Figure 4.15: Transfer send CamFlow event drops comparison figure

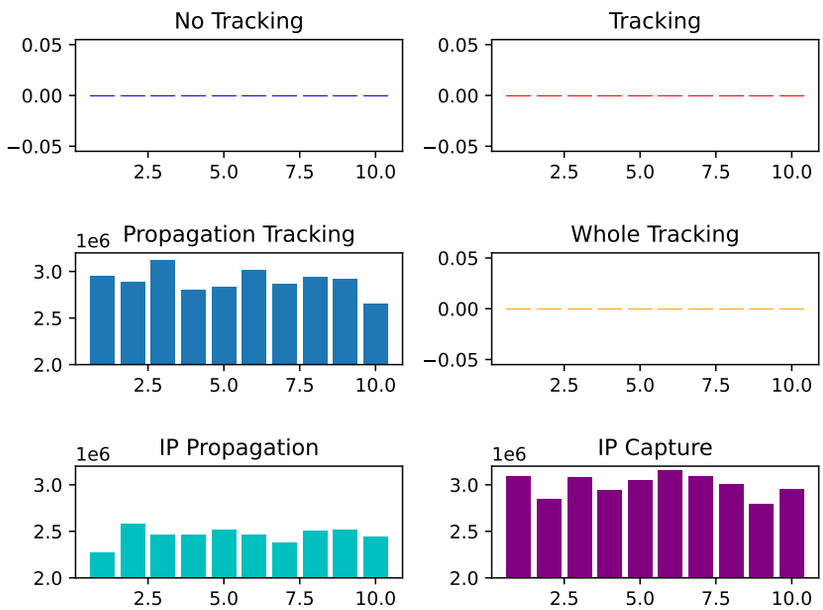

Figure 4.16: Transfer send CamFlow event drops individual figure

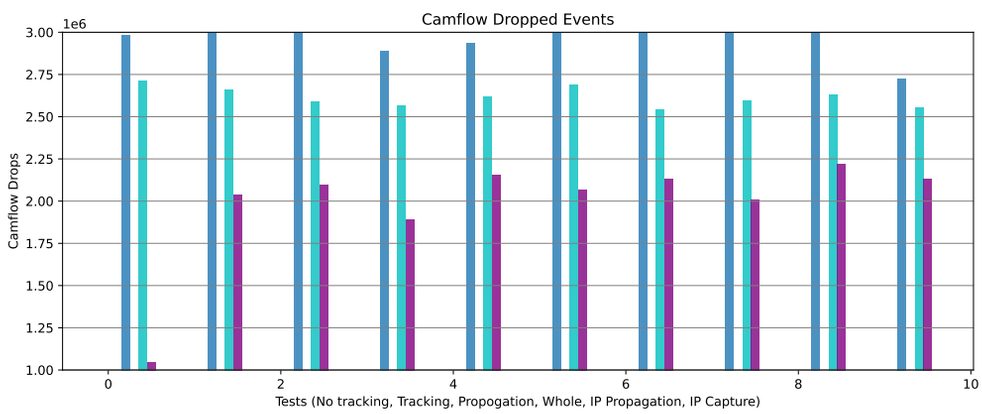

Figure 4.17: Transfer receive CamFlow event drops comparison figure

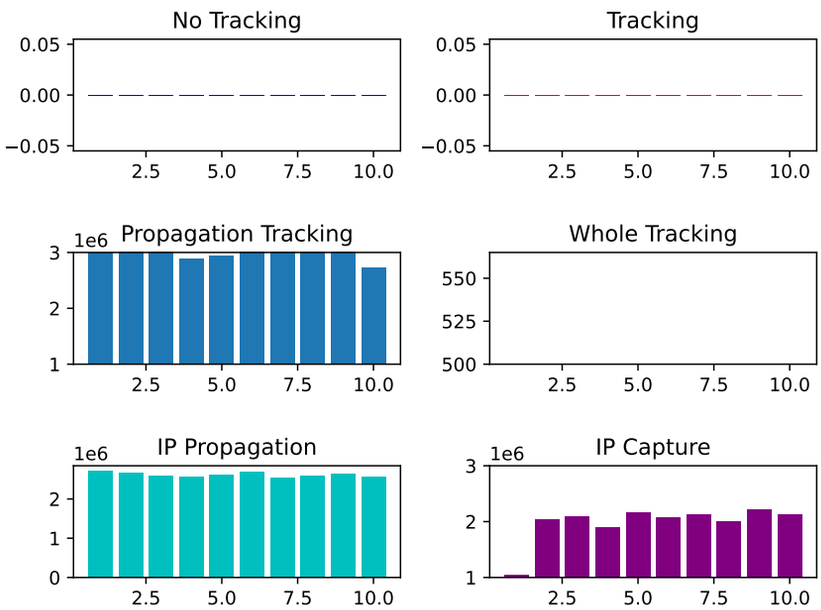

Figure 4.18: Transfer receive CamFlow event drops individual figure

Type	AVG. Messages	AVG. Time Sec	AVG. Size KB	AVG. CF-Drops
No Tracking Send	2999700	11.623142	0	0
No Tracking Recv	1119543.4	13.03359015	0	0
Track file Send	2999700	11.2652102	86.9	0
Track file Recv	1118935.9	12.70578647	89.7	0
Propagate file Send	2999700	13.75352838	402441.6	2900949.7
Propagate file Recv	1118529.4	14.96945362	426489.4	3027752.8
Whole system Send	2999700	11.60448034	16047.6	0
Whole system Recv	1119235.5	13.05461664	10084.7	0

Table 4.2: Tracking File of Messaging experiment

Type	AVG. Messages	AVG. Time sec	AVG. Size KB	AVG. CF-Drops
IP Propagate Send	2999700	22.77826242	586573.6	2461665
IP Propagate Recv	1117702.8	23.79365387	595493.3	2616868
IP Capture Send	2999700	24.12856894	569797.3	3003448.5
IP Capture Recv	1119713.6	25.16513004	688769.6	1978784.1

Table 4.3: Tracking IP of Messaging experiment

5

Discussion

5.1 Experimentation

As the system uses different underlying systems, most of the experiments were to test the components of Comm2Prov. Testing the components of the system individually gives a better overview of how they affect the system.

Due to the more widespread interest in MQTT, there are already several performance analysis done on MQTT protocol [23] [24] [25] out there. Hence, our experiments focused more on the performance of less evaluated components.

Due to the high configurability of Comm2Prov components, additional experiments could have been done testing different settings. Most of the experiments were to test the different provenance capture settings due to Comm2Prov's goal of configurable provenance. Even with several evaluation already being established on MQTT system, some experiments could have been conducted testing the different settings. Some of these settings are the different security options deploy-able on MQTT connections and brokers.

Something which could additionally be tested is MQTT provenance publishing of CamFlow. Here it would be interesting to see how much provenance data is published and it's speed of publishing to a MQTT broker. The different Qualities of Service that a message can be published on can also be compare to each other, testing the accuracy against performance issue. As a lot provenance data

is generated in short times, there is the worry if a MQTT broker is capable of keeping up with the rapid publishing. It is possible that published provenance messages are dropped by the MQTT broker because of the provenance generation. This aspect is something that would be interesting to evaluate further in order to see the applicability transferring provenance data over MQTT.

However, the capabilities of a MQTT broker is determined by its allocated processing power and network speed. Meaning that if the broker does not end up managing transferring provenance due to its rapid generation, the issue can be solved by allocating more processing, memory, and network capabilities to the broker. Testing the accuracy of insufficient computation power does not sound like an informative experiment, as the solution could easily be to add more computation. But there is a valid point of discovering how much computation power is needed to give a MQTT broker in order for it to keep up with the provenance generation. But this does require the elimination of other variables in order to get a clear image.

Virtualbox has some additional tools which could have been used to further enhance the experiments. Using the virtualbox interface, it is possible to monitor the network, CPU, and memory usage of the virtual machines running. Through the monitoring, the computation overhead caused by the capture mechanisms and the Comm2Prov provenance client could have been inferred from the data collected from monitoring. However, as it was in the case of the benchmark test, the monitor showed full CPU utilization while it was not tracked. This is to be expected, as the purpose of that experiment was to test capture mechanism on a stressed application. But it also sets the precedence that the monitor might not provide any particularly useful information.

5.1.1 Benchmark

As the benchmarking experiment were to test the provenance capture during high test computation, the test were done by saving the provenance to a local file. As the main part of Comm2Prov is for the provenance to be transmitted over MQTT network, the experiment can arguably be seen as less relevant to this thesis. However, with the uncertainty of the accuracy of provenance sent over MQTT as discussed earlier, evaluating the size of the provenance data generated could be less accurate.

Results

Figures 4.1-4.2 show the individual time results done by the experiment in graphs, while figures 4.3 - 4.4 show the auditing size, and figures 4.5 - 4.6 show

the drop rate of events. The area of what is displayed on these graphs have been configured to show the difference between each individual test. This is why some of the graphs with high difference in full comparison graph is less visible in individual comparison graph.

There were some outlier results coming out of the test, such as the propagation tracking test having two tests with significantly higher times than the others, being the first and ninth test. Other outliers can be that only the first test with propagation tracking and the seventh test of the whole system tracking had any sort of dropped events during the benchmark experiments. This is reflected in the audit log size captured during for the first propagation test, but not for the seventh whole system test. The cause behind these outlier results are yet to be determined.

These outliers where not excluded from the average result calculation, shown in 4.1, as all test iterations were used to run the test. This does affect the average result, leaving the table to display a slightly inaccurate end result due to possible irregular variables during testing. These irregular results could have been excluded from the average due to their unexplained appearance, but was refrained doing to better correspond to the graphs.

As seen in 4.1, there was no significantly noticeable difference between the different provenance capture settings. Most of the differences in time measured are minimal, with at most a minute of variation between the longest and shortest measured time. These variations can be due to small random differences occurring across the multiple machine learning calculations, stacking up to make the difference. The time variation could also be due unknown variables affecting the test, which can be related to the outlier results.

Seen in correlation with the amount of auditing data collected 4.3 - 4.4being consistent across the tests, the time was mostly unaffected by the provenance gathering mechanism. Despite tracking gathering 89.5 KB with provenance data, propagation gathering 265093.4 KB provenance data, and whole system tracking gathering 427109.9 KB, there were no significant variation caused by these times. It can be argued that there are some smaller increases in time consistent across tests 4.2, if you overlook some of the high spikes and drops. But these differences some seconds at most, a small increase in contrast to the difference of provenance data collected during the tests.

The average time calculated in the table 4.1 indicates that CamFlow provenance collection has small impact on systems already under high stress. Due to the the average time of the no tracking experiment is calculated to be higher than the regular tracking experiment, and the variations between each test vary within each test, making the overhead caused by CamFlow hard to induce by

these result. According to the performance tests done in the CamFlow paper[7], CamFlow has a 1%-11% overhead.

The experiment does show how much more provenance data is collected by the different capture settings4.3 4.4.

5.1.2 Networking

Due to an interaction between vagrant private network and CamFlow, IP-based tracking was configured in the CamFlow configuration file initiated through startup. This was caused by CamFlow requiring the IP they are to monitor to be either newly created or not yet established. Since vagrant sets up the private network during virtual machine start, the network connection is already established by the time experiments run. This does not affect the final result of the experimentation, as it could be carried out by resetting the virtual machine. But it reveals an inflexibility in the dynamic provenance configuration which Comm2Prov builds upon.

Both IP tests were done while tracking all incoming and outgoing packets for both sender and receiver. But capturing incoming packets on an address you know they are mostly sending messages to can end up recording unnecessary provenance data. Testing the difference between only capturing received messages on the receiver side and only capturing sent messages on the sender side can prune out potentially unnecessary logs.

Results

The graphs having with results are separated into the sender and receiver sides to see the difference in sending data and receiving data. As such, graph 4.9-4.10 - 4.7 - 4.8 represent the time measured in seconds, graphs 4.13-4.14 - 4.11 - 4.12 represent the audit log size collected in KB, and 4.17-4.18 - 4.15 - 4.16 represent the amount of dropped events.

Similarly to the benchmark experiment, there are some outlier results that seem irregular. These irregularities are still calculated in the average time, for the same rteasons as earlier. This spike in time occurs also on the first test of some of types of test in this experiment, being no tracking, regular tracking and IP tracking. This pattern of irregular time and occurrence can also be seen in the benchmark experiment to some extent. But the exact cause is still uncertain.

As seen in the comparing of time 4.7 4.9, provenance capture has a bigger

impact on the system as it tracks network packets. Tracking messages sent over network has a much bigger overhead than tracking configuration which primarily focus on the inner operating system. This is seen in the tables with average network transfer results 4.2 , with the average time and audit size being significantly bigger in propagation tracking, IP propagation tracking, and IP capture. This can be due to the network capabilities of the virtual machine outperforming the computation power of the system. This is further discussed in a later section 5.3, but the capture of packets seem to be causing noticeable overhead, which occur in propagation tracking, IP tracking, and IP Capture.

Graphs of the network transfer time 4.7 4.9 show that propagation tracking, IP tracking, and IP Capture usually use more time than the other configurations.

Occasionally, whole system provenance tracking uses more time than propagation tracking. The tracking configurations that track packets are the only configurations which has any CamFlow events drops, as seen in the graphs on dropped events 4.17 4.15 4.18 4.16.

IP packet capture test 1 goes beyond the upper limit set on the on the graph in the time comparison graphs 4.9 4.7, because would dwarf the other comparison if left to be in full view. The full view of them can be seen in the individual graph4.10 4.8, showing how much more time this singular test took in comparison to the other. Although seeming like an irregularity, it might not be. Comparing the time 4.10 4.8, audit log size 4.14 4.12, and dropped events 4.18 **??** of the first IP capture test, we can see that the IP capture test captured more events during this test than any others. From this, we can reason that the first IP capture test is closer to the true overhead caused by capturing packets, due to it recording almost every event captured. From another perspective, the irregularity might have slowed down the test and given more time for provenance data to be recorded. But even in this

5.2 Implementation

5.2.1 MQTT configured provenance

One of the big security factors is also the core concept of Comm2Prov, using MQTT to deliver provenance configuring commands. A part of MQTT is that the subscribers and publisher are essentially independent of one another. This causes publisher to be unaware to who gets their published messages, and subscribers to be unaware who publishes to the topics they subscribe too. This

puts MQTT perilous situation by nature of its protocol, as unprotected brokers and topics are prime target for Man-in-the-Middle or spoofing. Malicious actors with knowledge of how the system works can issue commands to turn on or off provenance if they get access.

Comm2Prov is not designed to be a open environment accessed by unknown actors, giving precedence for using the MQTT login approach to restrict actions and authorities. One common security measure implemented in MQTT brokers is the username and password security, creating a form of login upon connecting to a broker. Through this login, it could be possible to assign restrictions and permissions on users. CamFlow supports login structure, allowing it to used for simple authentication om the brokers side.

Applying stricter restriction on who can publish to which topics allows Comm2Prov instances to be further configured to only post on their own broker. Additionally, multiple brokers can be used with different security configurations to them. One broker may be used for issuing status of a provenance client, while another is used for issuing commands.

As stated in the survey done on MQTT by Biswajeeban Mishra and Attila Kertesz [26] , MQTT sacrificed encryption for the sake of being lightweight. It is due to this lightweight capability that MQTT was chosen as a component of our system, but in return ends giving it a security weakness.

The status list can be expanded to include more information, such as camflow capture state hash and remote attestation hash. Current status list of running or disconnected does not tell if provenance is active on system, only if it is connected.

5.2.2 CamFlow

CamFlow assumes that it is deployed in a trustworthy environment. Might not always be the case for different scenarios. CamFlow also does not encrypt the provenance data they publish over the MQTT. This issue remains in all forms that CamFlow can publish their provenance logs, not only over MQTT.

As stated in the paper by Dan Dinculeană and Xiaochun Cheng on Authorization on MQTT IoT devices[27], MQTT offers some security and authentication measures which may vary from MQTT systems. However, Camflow does not provide much for optional configuration to its MQTT provenance publishing. CamFlow provides the simple login option, with username and password, as the optional security measure. Adding more options requires a modification of CamFlow to fit the needs implemented on the MQTT broker. Moreover, this

username and password stored in the configuration file. This configuration file does require low-level privilege to access, and an assailant with access to this file can obscure their activities from provenance either way.

5.2.3 Comm2Prov System architecture and design

Comm2Prov is a system for enabling provenance and disabling provenance. This consequentially makes any provenance capturing system utilizing it only as secure as Comm2Prov. As Comm2Prov is based on CamFlow and MQTT, their security mechanisms decides some of the security Comm2Prov can enforce. But as CamFlow is open source and MQTT has open source implementations such as mosquitto[1] broker, security means can be further developed. This means that Comm2Prov can developed along with Camflow and a MQTT to fit the deployment's needs.

One of the strengths of using CamFlow is its ability to directly publish MQTT messages. A potential problem with developing a client to publish provenance that the published provenance data is caught by the capture and recorded again, causing an infinite loop of provenance generation. CamFlow fixed this by marking itself as an opaque entity which to not capture. A provenance publishing client could avoid the loop by using this same method. However, by not being forced to make the client an exception for capture, the client can potentially be captured. This could be useful in cases where one would want to capture how many orders the client receives and from which connections. But as expressed earlier, there are limitation to the security which can be applied to MQTT messaging provided by CamFlow.

5.3 CamFlow Provenance Data

As Comm2Prov can be deployed over multiple machines simultaneously, multiple machines can end up providing provenance in order to capture the flow of data. However, as the instance of CamFlow only knows about their own machine without any knowledge about a bigger network. Therefore, logs created by the transfer data experiment were inspected to learn what was captured in the different provenance levels and if they could be united to capture flow across networks. The provenance levels inspected were: regular process tracking, propagate tracking, and whole-system tracking. Some interesting discoveries were made from inspecting these graphs.

1. https://github.com/eclipse/mosquitto

Firstly, the captured provenance was parsed through SPADE [20] as input, where SPADE providing dot output to be used for graphs. Graphviz [28] was utilized to generate graphs of the provenance data. The provenance was recorded in the SPADE-JSON format to be processed by SPADE.

The graphs were made from the simple tracking, propagation tracking, and whole tracking provenance capture settings. Entities were set to be non-duplicate, making every transformation of the data its own entity within the graph. The network data experiment also reduced the amount of Lorem Ipsum paragraphs to a singular paragraph, in order to reduce the provenance logs size. Depending on the size of the logs, converting it to a graph can take from seconds to hours.

One discovery is that regular tracking and whole system tracking does not capture the packets sent over the network. Regular tracking does not expand outside of the process running, thus does not capture beyond what the process does. Whole-system tracking tracks all system calls to the kernel, registering that the process takes use of a socket, but does not register the packet which was transmitted. Propagate tracking managed to capture the packet sent from the sender, registering the length, recipient and sender of the packet. This makes regular tracking and whole system tracking less useful for the purpose of tracking data across networks. A possible reasoning for why whole-system tracking does not capture messages, is that it captures the socket's creation, but not what is transferred by using it the socket. Propagation tracking however tracks the data flow of data throughout a system, until it leaves the system, meaning it captures the messages sent over the socket. Reduced accuracy in this exact area is not a good sign for Comm2Prov, as the aspect of network transfer is a core point of Comm2Prov's concept.

Another discovery was made when the logs of both the receiver and sender of data were combined. The purpose of this was to see how close the provenance data of two independent instances of CamFlow provenance capture would get to each other. The raw data of these logs were combined before they were processed into graphs. This resulted in packets sent between the sender and recipient being referred to by both sides, meaning logs of independent instance can provide a connected logs over the network. Naturally, this only worked on provenance capture that registered packets, as they were the connecting point.

5.3.1 Future Work

There is much more work that can be done on Comm2Prov. One is that the Comm2Prov client could potentially be integrated into CamFlow as an ex-

tension to its existing functions. Additionally, using an open source MQTT broker as abase, a more complex and secure communication between CamFlow, Comm2Prov client, and the broekr can be further developed. There is also more optimization possibly in the source code, as well as implementing more optional commands and features, such as having the CamFlow policy has published on its status as it changes..

More and better experiments can also be done on the Comm2Prov system. A complete overhead measurement of Comm2Prov, testing of the accuracy and overhead caused by provenance being published to a MQTT broker. Additionally improvements can be done to try find the reasoning for the irregular results and assuring that they do not occur in future tests. The environment can also be further changed, testing on newer OS instances, as well as testing on virtual machines with access to less computing power.

6

Conclusion

Comm2Prov is a system showing the possibilities of distant auditing to IoT devices, allowing provenance to be configured as needed on IoT devices without direct access to them. With the distant selective auditing, an auditor can enable provenance on suspected IoT devices without their explicit awareness to gather data with only necessary amounts of auditing.

Experiments were conducted to test the viability of different capture configurations, to show and discuss the possible different usages of provenance distributed network system. First experiment tested the impact on provenance on local systems during high intensity computation, resulting in showing no significant impact by the capture. Another experiment was to test the capabilities of network capture, sending many messages in rapid succession over the network, resulting a small noticeable overhead in capturing network messages.

There were some unknown irregularities in the testing, causing some irregular result compared to the others. These irregularities seems to be mostly affected in their time measurement, giving some higher times than they should The result of these experiments showed that provenance capture can create large amounts of data in short time, potentially reaching gigabytes in a single minute or half an hour depending on settings.

There is potential for further development and experimentation on Comm2Prov, with a potential integration into CamFlow security model and testing of more configuration settings.

Citations

[1] Wajih Ul Hassan, Adam Bates, and Daniel Marino. "Tactical Provenance Analysis for Endpoint Detection and Response Systems." In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1172–1189. DOI: 10.1109/SP40000.2020.00096.

[2] Yulai Xie et al. "P-Gaussian: Provenance-Based Gaussian Distribution for Detecting Intrusion Behavior Variants Using High Efficient and Real Time Memory Databases." In: *IEEE Transactions on Dependable and Secure Computing* 18.6 (2021), pp. 2658–2674. DOI: 10.1109/TDSC.2019.2960353.

[3] Zijun Cheng et al. *Kairos: Practical Intrusion Detection and Investigation using Whole-system Provenance*. 2023. arXiv: 2308.05034 [cs.CR].

[4] Xueyuan Han et al. "FRAPpuccino: Fault-detection through Runtime Analysis of Provenance." In: *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*. Santa Clara, CA: USENIX Association, July 2017.

[5] Frank Wang, Yuna Joung, and James Mickens. "Cobweb: Practical Remote Attestation Using Contextual Graphs." In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. SysTEX'17. Shanghai, China: Association for Computing Machinery, 2017. ISBN: 9781450350976. DOI: 10.1145/3152701.3152705.

[6] Luc Moreau; Paolo Missier; Khalid Belhajjame; Reza B'Far; James Cheney; Sam Coppens; Stephen Cresswell; Yolanda Gil; Paul Groth; Graham Klyne; Timothy Lebo; Jim McCusker; Simon Miles; James Myers; Satya Sahoo; Curt Tilmes. *The PROV Data Model*. https://www.w3.org/TR/2013/REC-prov-dm-20130430/ [Accessed: 15.04.2024]. 30 April 2013.

[7] Thomas Pasquier et al. "Practical Whole-System Provenance Capture." In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Association for Computing Machinery. Santa Clara, California: Association for Computing Machinery, 2017, pp. 405–418. ISBN: 9781450350280. DOI: 10.1145/3127479.3129249.

[8] Md Morshed Alam and Weichao Wang. "A comprehensive survey on data provenance: State-of-the-art approaches and their deployments for IoT security enforcement." In: *Journal of Computer Security* 29.4 (June 2021), pp. 423–446. ISSN: 0926-227X. DOI: 10.3233/jcs-200108.

[9] Thomas Pasquier, David Eyers, and Margo Seltzer. "From Here to Provtopia." In: *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. Ed. by Vijay Gadepally et al. Cham: Springer International Publishing, 2019, pp. 54–67. ISBN: 978-3-030-33752-0.

[10] Gbadebo Ayoade et al. "Evolving Advanced Persistent Threat Detection using Provenance Graph and Metric Learning." In: *2020 IEEE Conference on Communications and Network Security (CNS)*. 2020, pp. 1–9. DOI: 10.1109/CNS48642.2020.9162264.

[11] Maya Kapoor et al. "PROV-GEM: Automated Provenance Analysis Framework using Graph Embeddings." In: *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2021, pp. 1720–1727. DOI: 10.1109/ICMLA52953.2021.00273.

[12] Weina Niu et al. "LogTracer: Efficient Anomaly Tracing Combining System Log Detection and Provenance Graph." In: *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*. 2022, pp. 3356–3361. DOI: 10.1109/GLOBECOM48099.2022.10000804.

[13] Mashal Abbas et al. "PACED: Provenance-based Automated Container Escape Detection." In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 2022, pp. 261–272. DOI: 10.1109/IC2E55432.2022.00035.

[14] Thomas F. J.-M. Pasquier et al. "Camflow: Managed Data-Sharing for Cloud Services." In: *IEEE Transactions on Cloud Computing* 5.3 (2017), pp. 472–484. DOI: 10.1109/TCC.2015.2489211.

[15] Thomas F. J.-M. Pasquier et al. "Information Flow Audit for PaaS Clouds." In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 2016, pp. 42–51. DOI: 10.1109/IC2E.2016.19.

[16] Sherif Akoush et al. "MrLazy: Lazy Runtime Label Propagation for MapReduce." In: *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. Philadelphia, PA: USENIX Association, June 2014.

[17] Kiran-Kumar Muniswamy-Reddy et al. "Provenance-aware storage systems." In: *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*. ATEC '06. Boston, MA: USENIX Association, 2006, p. 4.

[18] Adam Bates et al. "Trustworthy whole-system provenance for the Linux kernel." In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. Washington, D.C.: USENIX Association, 2015, pp. 319–334. ISBN: 9781931971232.

[19] Robert W. Wisniewski. "relayfs : An Efficient Unified Approach for Transmitting Data from Kernel to User Space." In: 2003.

[20] Ashish Gehani and Dawood Tariq. "SPADE: Support for Provenance Auditing in Distributed Environments." In: *Proceedings of the 13th International Middleware Conference*. Middleware '12. ontreal, Quebec, Canada: Springer-Verlag, 2012, pp. 101–120. ISBN: 9783642351693.

[21] Edited by Andrew Banks and Rahul Gupta. *MQTT Version 3.1.1*. Latest version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html [Accessed: 15.04.2024]. 29 October 2014.

[22] Randal S. Olson et al. "PMLB: a large benchmark suite for machine learning evaluation and comparison." In: *BioData Mining* 10.1 (Dec. 2017), p. 36. ISSN: 1756-0381. DOI: 10.1186/s13040-017-0154-4.

[23] Biswajeeban Mishra. "Performance Evaluation of MQTT Broker Servers." In: *Computational Science and Its Applications – ICCSA 2018*. Ed. by Osvaldo Gervasi et al. Cham: Springer International Publishing, 2018, pp. 599–609. ISBN: 978-3-319-95171-3.

[24] Davi L. de Oliveira et al. "Performance Evaluation of MQTT Brokers in the Internet of Things for Smart Cities." In: *2019 4th International Conference on Smart and Sustainable Technologies (SpliTech)*. 2019, pp. 1–6. DOI: 10.23919/SpliTech.2019.8783166.

[25] Malti Bansal and Priya. "Performance Comparison of MQTT and CoAP Protocols in Different Simulation Environments." In: *Inventive Communication and Computational Technologies*. Ed. by G. Ranganathan, Joy Chen, and Álvaro Rocha. Singapore: Springer Singapore, 2021, pp. 549–560. ISBN: 978-981-15-7345-3.

[26] Biswajeeban Mishra and Attila Kertesz. "The Use of MQTT in M2M and IoT Systems: A Survey." In: *IEEE Access* 8 (2020), pp. 201071–201086. DOI: 10.1109/ACCESS.2020.3035849.

[27] Aimaschana Niruntasukrat et al. "Authorization mechanism for MQTT-based Internet of Things." In: *2016 IEEE International Conference on Communications Workshops (ICC)*. 2016, pp. 290–295. DOI: 10.1109/ICCW.2016.7503802.

[28] Emden R. Gansner and Stephen C. North. "An open graph visualization system and its applications to software engineering." In: *Softw. Pract. Exper.* 30.11 (Sept. 2000), pp. 1203–1233. ISSN: 0038-0644.