A run-time framework for ensuring zero-trust state of client's machines in cloud environment

Devki Nandan Jha, *Member, IEEE*, Graham Lenton, James Asker, David Blundell, Martin Higgins, David C.H. Wallom

Abstract—With the unprecedented demand for cloud computing, ensuring trust in the underlying environment is challenging. Applications executing in the cloud are prone to attacks of different types including malware, network and data manipulation. These attacks may remain undetected for a significant length of time thus causing a lack of trust. Untrusted cloud services can also lead to business losses in many cases and therefore need urgent attention. In this paper, we present *Trusted Public Cloud* (TPC), a generic framework ensuring the *Zero-trust* security of client machine. It tracks the system state, alerting the user of unexpected changes in the machine's state, thus increasing the run-time detection of security vulnerabilities. We validated TPC on Microsoft Azure with Local, Software Trusted Platform Module (SWTPM) and Software Guard Extension (SGX)-enabled SWTPM security providers. We also evaluated the scalability of TPC on Amazon Web Services (AWS) with a varying number of client machines executing in a concurrent environment. The execution results show the effectiveness of TPC as it takes a maximum of 35.6 seconds to recognise the system state when there are 128 client machines attached.

Index Terms—Cloud Computing; Zero Trust; Privacy; Verification; Intrusion Detection; Trusted Platform Module

1 Introduction

In the past decade, cloud computing has evolved as an essential part of the computing paradigm. With numerous advantages including no upfront cost, pay-per-use, improved reliability and scalability, applications varying from enterprise management to healthcare have started to utilise cloud services for storage and computation purposes [1]. Typically cloud services are managed by a third-party organisation such as Amazon or Google with massive data centres geographically distributed all across the globe. The data centre resources are virtualised and offered to the clients depending on their requests.

There are a number of challenges in utilising the remote cloud environment. The most important of these challenges are those related to privacy and security [2], [3]. After a client uploads their data in the cloud, some aspects of control over that data are lost. They are required to trust the cloud provider and the system models under which different components of the cloud business operate [4]. Let us consider Infrastructure as a Service, where a service provider virtualises their physical resources to allow multiple clients to run heterogeneous applications in virtual machines. Within this type of environment, it is difficult to provide an absolute guarantees of privacy for each client's virtual machine or to prove the privacy of the resources that are being utilised.

In order to handle privacy and security in the cloud environment, it is important to understand the potential vulnerabilities a system may encounter. Various types of threats and attacks can be possible in the cloud environment including malware attacks, network attacks, data manipulation attacks and untrusted administrator [5]–[7]. On top of that, users and administrators can also inflict some security vulnerabilities in the underlying system. Operations such as leaked credentials, exposed network ports, wrong security patching and unresolved bugs can expose the system to various attacks. Detecting these vulnerabilities is hard and therefore may be left undetected for a long time. Research shows that the average time to detect a system compromise (Dwell Time) is currently 211 days for a middle-size cloud-based organisation. An addition of 67 days is required to contain it thus, leading to a business loss of on average \$4.8 million per middle-sized organisation[1].

To mitigate the security and privacy risks, numerous solutions are available in the literature [8]–[10]. These can be separated into *software-based* or *hardware-based* solutions. *Software-based* solutions e.g., antivirus or antimalware, regularly scan the system executables, comparing with a given threat database and isolate or remove applications identified as a threat. New security vulnerabilities are generated every day and it's hard for the software-based solutions to catch them. In addition to this, they are not able to verify their own authenticity. To overcome some of the issues, *hardware-based* solutions are proposed which leverage Trusted Platform Module (TPM) chip [11]. Since the chip is integrated with the system, the security enforced by the TPM is hard to break. However, processing using the TPM chip is very slow and it is not available in all cloud environments. An alternative is virtual/software TPM which is based on the TPM specifications and can recognise any system state change. However, they are not able to protect their own keys in case of a security attack [12]. Unfortunately, these traditional methods predominantly rely on perimeter-based defences i.e., they are mostly static and establish a trusted zone within the system/network and assume that entities inside this perimeter are inherently trustworthy. Given the growth of fast and dynamic applications in the cloud environment, traditional perimeter-based security becomes undesired.

- *D. N. Jha is with School of Computing, Newcastle University Newcastle Upon Tyne, UK and Oxford e-Research Centre, University of Oxford, UK. Email: dev.jha@ncl.ac.uk,*
- *G. Lenton, J. Asker & D. Blundell are with CyberHive Ltd., Newbury, UK. Email: {graham.lenton, james.asker, david.blundell}@cyberhive.com*
- *M. Higgins & D.C.H. Wallom are with the Oxford e-Research Centre, University of Oxford, UK. Email: {martin.higgins, david.wallom}@eng.ox.ac.uk*

Manuscript received ???; revised ???

1. https://www.ibm.com/uk-en/security/data-breach

Maintaining trust is one of the main concerns in the cloud environment. Recently, *zero-trust* security concept has been introduced which emphasises on the resource protection based on the principle that trust is never implicitly granted and must be continuously verified. However, most of the mainstream zero-trust architectures primarily focus on network security [2,3]. A few previous works have also addressed some related aspects of system security for zero-trust frameworks [13]–[16]. However, run-time detection of security threats is still a challenge. Also, it's hard to detect attacks that bypass security protocols and firewalls. Moreover, user/administrator-generated faults, which possibly lead to significant security risks, also need to be detected and resolved. In particular, this paper aims to answer the following research question:

How to detect if a system has been accidentally or maliciously altered at run-time?

To answer this question, we propose a novel framework, *Trusted Public Cloud* (TPC). It is based on the *zero-trust framework* which continuously tracks and analyses the client machine to determine if it is operating in either "$Trusted$" or "$Untrusted$" state. It is intended to be used by the local cloud administrator, such as the IT manager of an organisation. TPC framework stands out by offering robust protection against a wide range of threats, including those that manage to bypass conventional security protocols and firewalls. TPC 's strength lies in its ability to safeguard the system from internal threats that may arise due to actions by users or administrators, providing a layer of security that is often overlooked in the existing frameworks. Moreover, TPC can complement the existing network-specific zero-trust solutions and any such framework can be easily integrated.

We evaluated our TPC framework with an extensive real-cloud experiment to show it can detect any undesired activity in the application execution at run-time. We also evaluated the scalability of TPC to show the flexibility and efficacy for a *real production-ready* environment.

In summary, the main contributions of this paper are as follows:

- We have designed and developed a novel framework, *Trusted Public Cloud* (TPC) that analyses the trust state of a client machine at any instance of time using the zero-trust principle.
- A user interface is designed and implemented for the addition of a client machine to the TPC. It also allows the continuous run-time tracking of the client machine state.
- An extensive experiment is performed in the AWS and Azure cloud environment with the TPC components running in a cluster configuration.

The paper is organised as follows. Section 2 discusses the background and recent related works. Section 3 presents a formal model of the problem addressed giving the desirable properties of a proposed framework. Section 4 explains the architecture of TPC and discusses the detailed system activities while Section 5 presents the experiment evaluation. Before concluding the paper in Section 7, Section 6 gives some discussion and future works.

2. https://cloud.google.com/beyondcorp
3. https://www.paloaltonetworks.com/zero-trust

2 Background and Related Work

In this section, we review the recent related works on run-time monitoring, intrusion detection systems and trust state analysis.

2.1 Run-time Monitoring

Run-time system monitoring offers the live system state information including throughput, response time, CPU usage and memory usage. Monitoring the system metrics ensures that the service level agreements are met. It can also help in automating run-time scalability. Numerous works have been proposed by academia and industry to monitor the cloud system. [17]–[19] focuses on monitoring the run-time state of a cloud VM giving the system performance. [20], [21] adds the container performance along with the VM at any instance of time. [22], [23] are specific for monitoring the storage resources while [24], [25] focuses on the big-data system and illustrates the system performance. [26]–[28] are a few industry-based monitoring systems available to detect the system behaviour at run-time. Most of these systems collect the system performance information while ignoring the security information of the system.

A few works can also provide security-specific monitoring [7], [29]–[31]. In addition to this, PerSecMon monitors the combined performance and security information of the system [32]. Security monitoring requires kernel-level information to be collected at run-time. A kernel-module-based system is proposed in [33], [34] to gain the system insight while an advanced e-BPF-based system is proposed in [32], [35]. Although the given works analyse the system activity, they are not always able to detect the anomalies. Since an anomaly or attacker can manipulate system measurements, relying solely on system monitoring may not provide an accurate assessment of the system's state. Therefore, additional verification mechanisms are required to ensure the integrity of the monitoring process.

2.2 Intrusion Detection

The main purpose of Intrusion Detection in the cloud environment is to identify malicious behaviour at an early stage. The methods either detect a network-based intrusion or a host-based intrusion. Below is a summary of recent solutions presented to detect these threats.

Network-based intrusion Detection (NIDS). NIDS monitors the network traffic and safeguards against malicious requests. Traditionally, different heuristic algorithms such as Swarm intelligence, Artificial Bee algorithm, and Genetic Algorithms [36], [37] are used to detect such anomalies. Most of these methods match the request based on pre-defined anomalies or they try to find the odd one out among them. However, these methods are not able to detect new types of attack requests. To overcome this issue, many frameworks are proposed using machine learning and deep learning models [38], [39]. The frameworks are trained with the given anomalies data which is capable of detecting some unknown anomalies. The proposed frameworks commonly used Long Short Term Memory, Auto Encoders, Shapley Additive Explanations [40], [41]. A few works also propose to check the network traffic using Provenance-based methods [42].

Although the proposed methods are capable of detecting simple network intrusions, they are not able to detect encrypted packets transmitting through the network. Also, NIDS is not able to detect any internal attacks.

Host-based Intrusion Detection (HIDS). HIDS monitors the system logs to discover any intrusive activities. To discover host intrusion, various data mining and machine learning algorithms have been implemented. [43], [44] uses a classification and clustering approach while [45] uses the sequence time-delay embedding and hamming distance-based approach to find the anomaly. These methods monitor the system calls and generate some alarming messages when it detects any abnormal activity. Since the system metrics are generated for every single process execution, it is hard to manage such a large set of metrics. Moreover, the captured system metrics are not robust in case of a persistent threat as the intruder can manipulate the system logs to make the system appear uninterrupted [46], [47].

2.3 Trust State Analysis

Establishing trust in the cloud environment is a major concern for users. Multiple frameworks have been proposed in past to analyse the trust state of the cloud. [48], [49] uses the monitored security measurement metrics to specify the trust state. A third-party evaluation system is used in [50]–[52] where either an independent evaluating system or a consumer-based system is used. Trusted Platform Component (TPM) secure booting can also help in guaranteeing the trusted state [53], [54]. These systems rely on the trusted computing infrastructure (e.g., TPM, TXT) to verify the measurements generated by the system. However, the public cloud lacks a physical TPM. Software TPMs[4] can be available but the secret key of software TPM can easily be manipulated thus lacking a concrete trust guarantee.

2.4 Zero-Trust Frameworks

Recently, there has been an increase in zero-trust architecture (ZTA) from industry such as Google BeyondCorp[5], Palo Alto Zero Trust[6] and Cisco Zero Trust[7] which offer organisations a more practical and step-by-step approach for application deployment. However, most of these existing ZTAs have primarily concentrated on network security with limited efforts directed towards implementing a zero-trust framework for system security. A few previous works have also addressed some related aspects of system security for zero-trust frameworks. [13], [14] offers device and connection security for IoT systems. [15] propose a zero-trust framework for mobile-edge computing where a new component/user is validated using fingerprint and blockchain. [16] also utilises a zero-trust framework to dynamically control user's permissions to provide endpoint security. [55] propose S-ZAC to offer an access control mechanism for service mesh-based solutions in the cloud using SGX. However, none of the proposed frameworks consider the threats posed by the internal processes or users.

3 System Overview

In this section, we first present the system description and formal definition of trust definition in the cloud environment (Section 3.1). Using the formal model, we define our problem and explain the desirable properties in a proposed solution (Section 3.2).

4. https://github.com/stefanberger/swtpm
5. https://cloud.google.com/beyondcorp
6. https://www.paloaltonetworks.com/zero-trust
7. https://www-cloud.cisco.com/site/us/en/solutions/security/zero-trust/index.html

3.1 Formal Model

Let C be a cloud service provider offering client machines $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3, ...\}$. The client machines can either be a virtual machine ν or a container κ i.e., $\mathcal{V}_i \in [\nu, \kappa]$. Each client machine $\mathcal{V}_i$ is employed by a user $\mathcal{U}_j$ where $\mathcal{U}_j | j = \{1, 2, ...\} \in \mathcal{U}$. A user $\mathcal{U}_j$ executes α set of applications $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_\alpha\}$ on the employed client machine $\mathcal{V}_i$. The execution of an application $\mathcal{A}_k$ interacts with a set of files $F_k \in F$. F can be a collection of user files $F^\mu = \sum_n f^\mu_n | n = \{1, 2, ...\}$ and system files $F^s = \sum_m f^s_m | m = \{1, 2, ...\}$ i.e., $F = F^s \cup F^\mu$. The interaction of an application and the files creates an injective mapping $\mathcal{A}_j \rightarrow F_o$ where F_o is a subset of F and is given as $F_o = \sum_{l_1, l_2} (f^\mu_{l_1} \cup f^s_{l_2}) \subseteq F$.

Consider a security vulnerability $v_k \in V$ caused by an entity $\mathcal{E}_l \in \mathcal{E}$. The entity $\mathcal{E}_l$ can be caused by either an external entity $\mathcal{E}^{ex}$ or an internal system user $\mathcal{E}^{in}$. The vulnerability V affects the execution of an application $\mathcal{A}_j$ in undesired ways. It either tries to read or update the system and/or user files F_o. Consider $F_v \subseteq F_o$ represents a set of files affected by the vulnerability. Finding F_v can help in finding the vulnerabilities.

Based on whether a vulnerability is present in the system, the state of a client machine $\mathcal{V}_i$ is defined as trusted $\mathbb{T}$ or untrusted $\mathbb{U}$ i.e., $\mathcal{S}(\mathcal{V}_i) \vdash \mathbb{T}$ and $\mathcal{S}(\mathcal{V}_i) \vdash \mathbb{U}$ respectively. A client machine $\mathcal{V}_i$ is defined as trusted if there is no known vulnerability $v_k \in V$ found at any time t as given in equation 1. Similarly, a client machine $\mathcal{V}_i$ is said to be untrusted, if there exist one or more known vulnerabilities $v_k \in V$ as given in equation 2.

$$\mathcal{S}(\mathcal{V}_i) \vdash \mathbb{T} \iff \forall t (\nexists v_k) \tag{1}$$

$$\mathcal{S}(\mathcal{V}_i) \vdash \mathbb{U} \iff \exists t (\exists v_k) \tag{2}$$

Finding the state of a client machine $\mathcal{S}(\mathcal{V}_i)$ at any time instance t to know whether the system is trusted is challenging.

3.2 Desirable Properties

The problem addressed in this paper is to find all K vulnerabilities $\sum_{k=1}^{K} v_k$ in the client machine $\mathcal{V}_i$ in a minimal time. The proposed framework needs to have the following desirable properties:

- *Exhaustive system monitoring*: The proposed framework is desired to capture the system measurements comprehensively. In addition to this, the captured measurements need to have detailed information about the process and files.
- *Adaptive monitoring*: The proposed system needs to be adaptive as all the files are not equally important. Since monitoring can generate a huge amount of data that need to be evaluated, setting an adaptive scheme constrains the volume of generated data thus making the evaluation faster.
- *Chain of trust*: There is some case scenario where an attacker changes the value of a file from say A to B and then at sometime later loads A, making it look like it is unchanged. Monitoring the file value only makes it hard to recognise the intrusion. Thus it is important to keep a chain of trust so that any change can be easily identified.
- *Fast*: Finding any vulnerability affecting the client machine in a minimal time is a key requirement of the proposed framework.

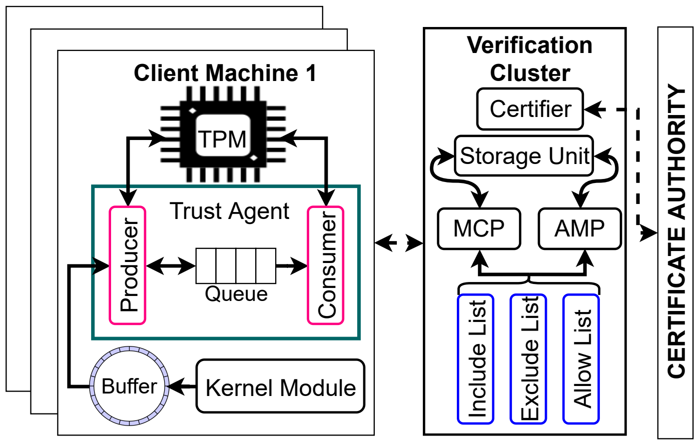


Fig. 1: System Architecture of TPC.

```
{
   "index":1,
   "dig":"fcf3b6e290b9ca63fad2262e39e7e6aed6f7badd1ba059fdc4b9
        caa063d29dd6",
   "fh":"f110ab97b9fcd04eb9da6ad67676aaa505cad1adff53a1768e0b9
        2e09c2d7250",
   "ino":2894068,
   "pid":2601,
   "pname":"cat",
   "ppid":2588,
   "ppname":``bash",
   "uid":0,
   "euid":0,
   "gid":0,
   "egid":0,
   "ts":"2023-3-09T16:54:37.697",
   "filename":"/opt/demo/test.doc"
}
```

Fig. 2: A sample measurement collected by the *Kernel Module*

4 Proposed Approach

Given the desirable properties, we propose and implement TPC. This section discusses the architecture and the implementation details of TPC.

4.1 TPC Architecture

TPC consists of 4 main components a) *Kernel Module*, b) *Trust Agent*, c) *Trusted Platform Module* and d) *Verification Cluster*. The first three components are executed on all connected client machines while the *Verification Cluster* is a centralised server communicating with the client machines. Figure 1 illustrates the schematic architecture of TPC and the dependencies among various components. The details of each component are given below.

4.1.1 Kernel Module

Kernel Module component monitors the system and captures the measurements. The measurements are represented in the form of an extended Integrity Measurement Architecture (IMA) list [53]. The extended IMA list captures detailed information about all the files and processes executing on the system. The measurements are stored in a circular memory-mapped buffer. When a file is opened for reading/updating, the kernel generates a hash of the file contents which is also added to the buffer. Figure 5 show a sample measurement captured by the *Kernel Module*. As shown in Figure 5, a measurement consists of an index, inode, process id, process name, parent process details, used details, timestamp with file hash and digest value.

4.1.2 Trust Agent

Trust Agent is the main component of TPC. It reads the measurements from the buffer and interacts with the *TPM* and *Verification Cluster*. The first operation performed by *Trust Agent* is to enrol the client machine with the *Verification Cluster*. The details of enrolment are given in Section 4.2.1. After the successful enrolment, it initiates the *Kernel Module* and starts up the *TPM*.

It has three elements *Producer*, *Consumer* and *Queue*. Both *Producer* and *Consumer* are asynchronous components accessing the central *Queue*. The *Producer* retrieves the measurements from the buffer and creates a measurements batch. It also communicates with the *TPM* to get the digest value of each measurement and populate the "*dig*" field with the *TPM* obtained digest value. The *Consumer* accesses the measurement batch and adds a TPM signature. The batch is later sent to the *Verification Cluster* for evaluation. This process is repeated continuously.

4.1.3 Trusted Platform Module (TPM)

TPM is a cryptographic co-processor chip included on almost every enterprise personal computer, server, and laptop motherboard. However, it may not be available for the public cloud's virtual machine where a *software TPM/virtual TPM/SGX-embedded TPM* can emulate similar behaviour. An industry consortium, Trusted Computing Group (TCG) provides the specification of the *TPM*. TPC uses *TPM* to perform two main functions, a) creating a digest for the measurement and b) performing the signature operation on the measurement batch. For each measurement, Platform Configuration Register (PCR) is used to create a digest using `pcrextend`. PCR store cryptographic hashes of system measurements to ensure the integrity and secure boot of the platform. The updated PCR value is read using `pcrread` and the obtained value is returned to the *Producer* to be added to the measurement's "*dig*" field. Similarly, for the given batch, *TPM* uses the *key handle* to create a signature. The signature is returned to the *Consumer* and is appended to the batch. The functionalities of *TPM* are accessed using the TPM 2.0 TSS Enhanced System API (ESAPI)[8].

4.1.4 Verification Cluster

Verification Cluster enrols a client machine and verifies the measurement received from the enrolled client's machine. It has 5 elements working together to evaluate the client machine's status. The first element is a collection of three lists namely: *Include List*, *Exclude List* and *Allow List*. *Include List* and *Exclude List* contain the system directory to be included and excluded for verification respectively. Since, files and processes are typically organised within system directories e.g., $/dev$ or $/usr$, TPC provides users with the flexibility to manage which directories are prioritised for verification through the use of an *Include List* and an *Exclude List*. These lists allow users to specify which directories should be included or excluded from the verification process, thereby indirectly controlling the evaluation of the files and processes within those directories. This ensures that the monitoring process is adaptive and can be tailored to prioritise the most critical files and processes

8. https://github.com/parallaxsecond/rust-tss-esapi

while excluding those seem less important. *Allow List* contains a list of file hash to validate the measurements. The *file hash* is computed at the client machine's boot time and stored in the *Allow List*. The hash of measurement files is always matched with the hash values stored in the *Allow List*. The *Measurements Configuration Processor (MCP)* manages and configures these three lists. The lists are created and associated with the client machine at the time of enrolment. An administrator can also update these lists at run-time in case of any legitimate change happened on the client machine.

Storage Unit stores the measurements received from the client's machine. The measurements can be parsed and retrieved by querying the storage unit. *Allowed Measurements Processor (AMP)* is the component which processes the batches of measurements to determine the trust status of the client machine. The *AMP* not only compares the file hash but also validates the measurement digest and signature to check if the measurements are valid. The verification process details are given in Section 4.2.3. To bring an extra layer of trust, the *Certifier* provides a unique digital certificate for each client machine. The certificate is generated with the help of an external *Certificate Authority*.

A User Interface is available to interact with the *Verification Cluster*. An administrator can easily manage the client machines where a machine can be enrolled, updated, or deleted. An administrator can also view the details of a client machine including the measurements' details, timestamp and client machine's state.

Verification Cluster can handle multiple client machines. Since the *Verification Cluster* is inherently deployed across multiple availability zones, there is no single-point failure.

4.2 System Activities

This section describes the system activities performed by TPC.

4.2.1 Enrolment

Each *Client Machine* $\mathcal{V}_i$ needed to be enrolled with the *Verification Cluster* VC prior to the verification process. In the beginning, an administrator creates a machine ID (id) and a token (tk). The id, tk and the *Verification Cluster* address are sent to the *Client Machine* for later use to communicate. The *Client Machine*, $\mathcal{V}_i$ configures the *Trust Agent* with the given id and performs an enrolment process with the *Verification Cluster*. This process enforces trust between the *Client Machine* and *Verification Cluster*.

The *Client Machine* first checks whether the system is already enrolled or not. If it is not enrolled, the *Client Machine*'s private key $\rho(\mathcal{V}_i)$ is used to generate a certificate signing request (CSR) $\sigma(\mathcal{V}_i)$. The CSR is then sent to the *Verification Cluster* requesting a certificate. First, the *Verification Cluster* extracts and saves the public key $\varrho(\mathcal{V}_i)$ from the CSR. The *Verification Cluster* is associated with a public Certificate Authority CA. After receiving the CSR, *Verification Cluster* sent it to the CA which generates and returns a certificate $\xi(\mathcal{V}_i)$. *Verification Cluster* saves a copy of the certificate before sending it back to the client machine. The pseudo-code for the enrolment process is given in Algo 1.

9. ⇒: Remote communication, →: Internal communication

Algo 1: Client enrolment process

```
Input: $\rho(\mathcal{V}_i)$ - Private key of the client machine $\mathcal{V}_i$, $VC$
       - Verification Cluster, $CA$ - Certificate Authority
Output: $\xi(\mathcal{V}_i)$ - certificate
1  //Check if the Agent Machine is already enrolled
2  if $\xi(\mathcal{V}_i)$ = NULL then
3      // Generate Certificate Signing Request (CSR) with the Client
       Machine's Private Key
4      $\rho(\mathcal{V}_i) \xrightarrow{generates} \sigma(\mathcal{V}_i)$ 9
5      // Send CSR to the Verification Cluster
6      $\mathcal{V}_i \xRightarrow{\sigma(\mathcal{V}_i)} VC$
7      // Verification Cluster extracts public key from the CSR and
       saves a copy
8      $\sigma(\mathcal{V}_i) \xrightarrow{extracts} \varrho(\mathcal{V}_i)$
9      // Request Certificate Authority to get the Signature
10     $VC \xRightarrow{\sigma(\mathcal{V}_i)} CA$
11     $CA \xRightarrow{\xi(\mathcal{V}_i)} VC$
12     // Save a Copy of the Signature and send it to the Client
       Machine
13     $VC \xRightarrow{\xi(\mathcal{V}_i)} \mathcal{V}_i$
14 end
```

4.2.2 Measurements' Digest and Signature Generation

After the successful enrolment of a client machine $\mathcal{V}_i$, the *Kernel Module* monitors and captures the system measurements $\mathcal{M}_s|s \in \{1, 2, ...\}$. For each measurement $\mathcal{M}_s$, a digest $\mathcal{M}_s^{dig}$ is generated by the *TPM*. The TPM uses *pcrextend* to generate a digest value for the selected *PCR* as given in equation 3. Here, $||$ represents a data concatenation process. The digest $\mathcal{M}_s^{dig}$ is read using TPM's *pcrread* for the selected *PCR* and is appended to the measurement as given in equation 4.

$$PCR_{new} \leftarrow pcrextend(PCR_{old}||\mathcal{M}_s) \tag{3}$$

$$\mathcal{M}_s^{dig} = pcrread(PCR) \tag{4}$$

After collecting a set of measurements, a batch $\mathcal{B}_l = \sum_{s=0}^{\Gamma} \mathcal{M}_s$ is constructed. A TPM signature $\mathcal{B}_l^{sig}$ is computed and added to the batch before sending the batch to the *Verification Cluster*. The signature uses the TPM's internal signature key as shown in equation 5. The whole process is repeated for all the measurements asynchronously.

$$\mathcal{B}_l^{sig} = sign(\mathcal{B}_l) \tag{5}$$

4.2.3 Verification

Each *Client Machine* has a state $\mathcal{S}(\mathcal{V}_i)$ which is either Trusted $\mathbb{T}$ or Untrusted $\mathbb{U}$. The state is determined by a comparison of the measurements submitted by the client's machine $\mathcal{V}_i$. The initial state is always Trusted and the later state is determined depending on the upcoming measurements. The verification process is divided into three parts as given below:

1) Verify the validity of *Client Machine*: The first step of verification is to check whether the *Client Machine* is already registered. The *Verification Cluster* checks the machine ID (id) and the certificate ($\eta(\mathcal{V}_i)$). Failure of matching these leads to irrecoverable untrusted status $\mathcal{V}_i$.

Algo 2: Verification process

```
Input: $\mathcal{B}_l \in \mathcal{B}$ - Batch of measurements submitted by
       the Client Machine $\mathcal{V}_i$,
Output: $\mathcal{S}(\mathcal{V}_i)$ - State of the Client Machine $\mathcal{V}_i$
// Verify the validity of Client Machine
Extract $id$ and $\eta(\mathcal{V}_i)$ from $\mathcal{B}_l$
if ! ($id$ & $\eta(\mathcal{V}_i)$ stored) then
    exit (irrecoverable error)
end
// Verify the crypto operation
Extract $\mathcal{B}_l^{sig}$ from $\mathcal{B}_l$
if !Verified($\mathcal{B}_l^{sig}$ using $\varrho(\mathcal{V}_i)$) then
    Set $\mathcal{S}(\mathcal{V}_i)$ to $\mathcal{U}$
    Exit (irrecoverable error)
end
for $\forall \mathcal{M}_s \in \mathcal{B}_l$ do
    Extract $\mathcal{M}_s^{dig}$
    if !Verified($\mathcal{M}_s^{dig}$) then
        Set $\mathcal{S}(\mathcal{V}_i)$ to $\mathcal{U}$
        Exit (irrecoverable error)
    end
end
// Verify the measurements hash
for $\forall \mathcal{M}_s \in \mathcal{B}_l$ do
    if ($\mathcal{M}_s \in$ Include List & $\mathcal{M}_s \notin$ Exclude List) then
        Extract $\mathcal{M}_s^{hash}$
        if !exists($\mathcal{M}_s^{hash}$ in Allow List) then
            Set $\mathcal{S}(\mathcal{V}_i)$ to $\mathcal{U}$
            Add a flag $\mathcal{F}$ to revisit $\mathcal{M}_s$
        end
    end
end
```

2) Verify the crypto operation: If the *Client Machine*'s validity is satisfied, the next step is to verify the cryptography operations performed by the TPM. First, the measurements batch $\mathcal{B}_l$ is parsed and the signature $\mathcal{B}_l^{sig}$ is extracted. The saved public key $\varrho(\mathcal{V}_i)$ during the enrolment process is used to validate this. If the validation is successful, the measurements batch is parsed further and each measurement is evaluated. For each measurement $\mathcal{M}_s$, the digest $\mathcal{M}_s^{dig}$ is extracted and verified. Failure to verify either signature $\mathcal{B}_l^{sig}$ or digest $\mathcal{M}_s^{dig}$ leads to making the *Client Machine* untrusted. Since the digest $\mathcal{M}_s^{dig}$ guarantees a chain of trust, failure to verify the digest or signature leads to irrecoverable untrusted status.
3) Verify the measurement hash: Once the cryptography operations are validated, the measurements are filtered according to the associated *Include List* and *Exclude List*. The hash value of each measurement $\mathcal{M}_s$ in the *Include list* is compared with the *Allow List*. If the measurement fails to satisfy, a flag is associated with the measurement and the *Client Machine* is made Untrusted. This type of failure is recoverable as it also pings the Administrator to verify the cause of failure. The Administrator can verify the reason of failure and either make it Trusted or leave it Untrusted.

The pseudo-code for the verification process is given in Algo 2.

Figure 3 shows the state transition diagram of a *Client Machine*. The initial state of the client machine $\mathcal{S}(\mathcal{V}_i)$ is always Trusted. The state $\mathcal{S}(\mathcal{V}_i)$ remains Trusted if the measurements are validated (transition 1). If the changes do not match, the machine state is set to Untrusted and notifications are sent to the administrator (transition 2). The reason for the failure is then analyzed. If the issue is due to the validity of the client or a crypto operation, the machine state remains Untrusted (transition 3). However, if the analysis determines that the changes are valid, the machine state is updated to Trusted (transition 4). .

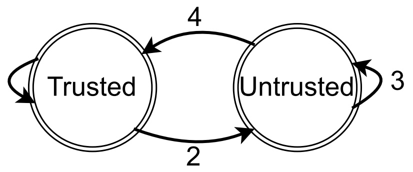


Fig. 3: *Client Machine* state transition diagram.

4.3 Implementation

Trust Agent is implemented in *Rust* while the TPM is accessed using *Rust* implementation for TPM 2.0 TSS ESAPI. *Kernel Module* is primarily implemented in *C*. Both *Trust Agent* and *Kernel Module* are packaged independently and made available using *apt-get* to be installed on the client's machine.

Verification Cluster is presented as an independent service and is mainly implemented in *Python*. The *Storage Unit* of *Verification Cluster* is implemented using *PostgreSQL*, *OpenSearch* and *Kafka*. The *MCP* and *AMP* components of *Verification Cluster* are packaged in Docker containers to be deployed independently. All the communications are performed using Restful APIs.

5 Experiment and Analysis

In this section, we evaluate TPC for both performance and scalability. The main focus here is to detect the machine's state which may be changed by the user/administrator or an anomaly which remains undetected for a very long time.

5.1 TPC Performance Evaluation

5.1.1 Environment Setup

To evaluate the performance of TPC, we deployed the project on Microsoft Azure. The VM configuration is *Standard DC2ds v3* with 2 vCPUs, 16 GiB RAM and 150 GiB storage. The VMs are enabled with Intel SGX's confidential computing features[10]. The VM is installed with *Ubuntu 20.04 LTS* with *linux-image-5.15.0-1037-azure* Kernel.

The *Kernel Module* and *Trust Agent* packages are installed from an *apt-get* repository. The *TPM* is made available in 3 forms, a) Local, b) Software TPM (SWTPM) and c) Software Guard Extenyion based SWTPM (SGX-SWTPM). Local is implemented with the help of Rust *OpenSSL* library. To use SWTPM, we leveraged Stephen Berger's SWTPM [11] which is based on LibTPMS and provides TPM emulators over the

10. https://learn.microsoft.com/en-gb/azure/virtual-machines/dcv3-series
11. https://github.com/stefanberger/swtpm

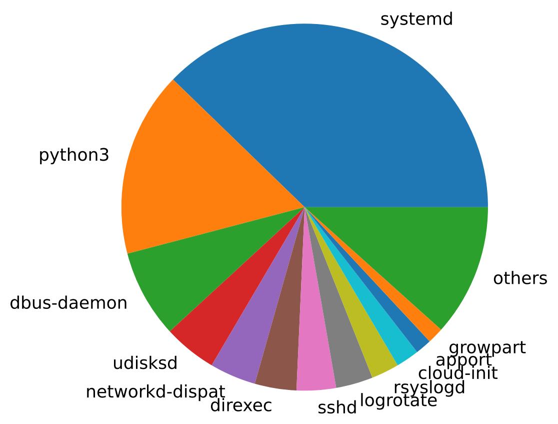


Fig. 4: Pie chart showing the set of processes executed at the given time.

Socket connection. In the SGX-SWTPM case, SWTPM is implemented inside Intel SGX using Occlum[12]. Occlum allows unmodified applications to run inside SGX. All the dependencies for *TPM* are pre-installed in the client machine before starting the experiments.

The *Verification Cluster* is executed on the AWS cloud environment where each component is executed independently. The OpenSearch, Kafka and PostgreSQL components of the Storage Unit as discussed in Section 4.3 are executed on *Amazon OpenSearch Service*, *Amazon MSK* and *Amazon RDS for PostgreSQL* respectively. The *MCP* and *AMP* components are executed in *Amazon EKS*. The execution is performed in a fixed-size environment with minimal scaling, to allow for the examination of individual components under workload stress (*Staging Cluster*).

Workload Settings. To evaluate the project, we wrote a test program that can create a set of files, edit and adds some content, reads the value and finally deletes the file. A test *Include List* containing a list of directories and a test *Allow List* containing 100 file contents and respective hash values are created. The program is made available as a service that starts with the system reboot. The whole setup is executed for 2 hours and the results are evaluated.

5.1.2 TPC *Performance Results*

Monitoring Results. TPC is able to monitor the system performance and capture the details of any processes executing at a given time. Figure 4 demonstrates that the framework effectively captures and reports detailed information about all executing processes at any given time. At the test time, a total of 490 processes are executed on the system. As the figure shows `systemd` covers the highest area with a total of 37.5 % processes followed by `python` at 16.3 %.

TPC also captures the details of each process which can be visualised in different formats (JSON, YAML or CSV file). The detail of a process in JSON format is given in Figure 5. The process details represented here extend the measurements captured from the *Kernel Module* by adding numerous details including client machine information, verification status as well as verification time. Figure 6 shows the total number of processes executed during the experiment duration.

12. https://github.com/occlum/occlum

```
{
    "_index": "measurements-2023-06-01",
    "_type": "_doc",
    "_id": "-FKfdogBC2KXPF7qOWRg",
    "_version": 1,
    "_score": null,
    "_source": {
        "dig": "47c582ba5658a19ab8c01d75b51e28f8f71e44c3c776c9
            9cab0c793a67426f4c",
        "egid": 1000,
        "euid": 1000,
        "fh": "96be05721a72b6cf024a37ad646a5076b9a39137669603d
            39ccde5e4d5b2206c",
        "filename":
"/home/azureuser/cyberhive/fad/4483e8fe-275c-4610-9f30-db70568
     fcc12/613_of_997_0069d235-6a9f-43f5-a92f-64db9513979e",
        "gid": 1000,
        "index": 82195,
        "ino": 541924,
        "pid": 8718,
        "pname": "python3",
        "ppid": 1,
        "ppname": "python3",
        "ts": "2023-06-01T11:02:00.085Z",
        "uid": 1000,
        "batch": "ef0055c0-7d20-4c6f-acc4-fa8034b54c09",
        "sequence": 12,
        "amp_matched": true,
        "eval_ts": "2023-06-01T11:02:06.338429",
        "eval_id": "083028e9-77b0-4135-96c2-9b2f8efa3933",
        "machine": "fc1f8541-536d-4335-bc84-214fe01af7c8",
        "amp_config_id": "927f0c02-bfe0-4219-a8ad-47d0cd582bed
            ",
        "amp_config_type": "includelist"
    },
    "highlight": {
        "machine": [
"@opensearch-dashboards-highlighted-field@fc1f8541-536d-4335-
     bc84-214fe01af7c8@/opensearch-dashboards-highlighted-
     field@"
        ]
    },
    "sort": [
        1685617326338
    ]
}
```

Fig. 5: Details of a measurement in JSON after verification process in the *Verification Cluster*

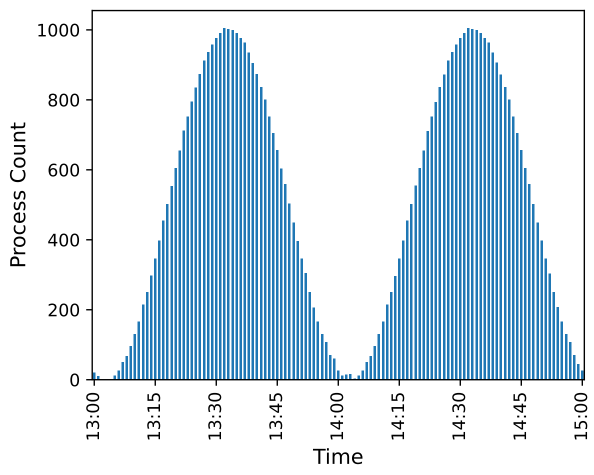
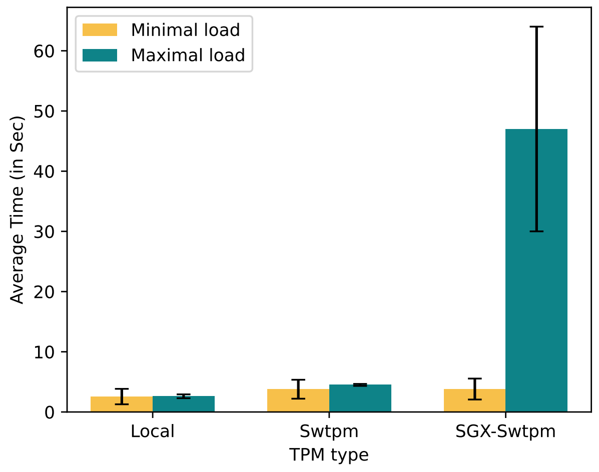
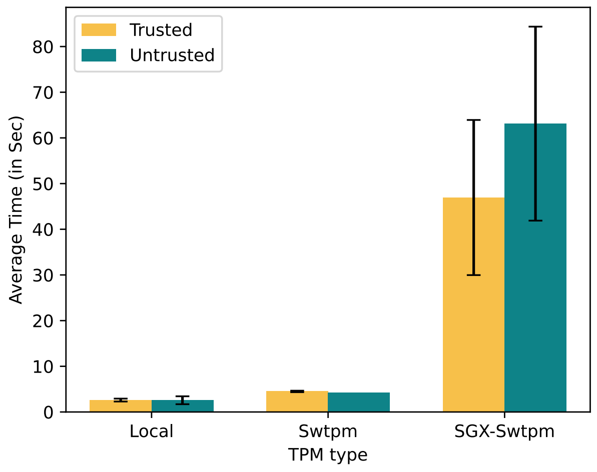


Fig. 6: Number of processes executing for the experiment duration.

Verification Time. Figure 7a shows the average time to verify the client machine in a minimal and maximal load condition while varying the *TPM* types (Local, SWTPM and SGX-SWTPM). For the minimal load scenario, we observed that all three cases exhibited similar processing times, 2.56, 3.78 and



(a)



(b)

Fig. 7: Figure showing the average time to verify the client machine in (a) varying load conditions (b) for *trusted* and *untrusted* measurements

3.81 seconds for Local, SWTPM and SGX-SWTPM respectively. However, under maximal load conditions, we noticed that Local and SWTPM demonstrated comparable processing times of 2.60 and 4.49 seconds respectively, while SGX-SWTPM exhibited a higher processing time of 47.010126 seconds. This increased processing time in SGX-SWTPM can be attributed to the queuing delay as the hashing and signing procedures, in this case, involved transferring the data to SGX, necessitating multiple layers of data transfer.

It is important to emphasize here that although SGX-SWTPM had the highest processing time, it is the most secure method as none of the data is openly accessible while the system is under execution. This highlights the trade-off between processing time and data security, wherein SGX-SWTPM prioritizes enhanced security measures over faster processing. Additionally, it is worth noting that the maximal load condition, where the processing time differences were observed, is not a common occurrence. While this condition may not be prevalent in practical scenarios, it is still relevant to explore and understand its implications for the system's performance. In future work, we are planning to have a nested *TPM* execution using SGX-SWTPM with Local or SWTPM to provide a similar range of security in an effective time period.

We conducted an additional analysis comparing the trusted and untrusted sub-cases within the three main cases. As shown in Figure 7b, similar to the previous findings, we found that SGX-SWTPM exhibited a higher processing time due to the reasons mentioned earlier. On the other hand, the processing times for the trusted and untrusted sub-cases in cases Local and SWTPM were comparable.

An interesting observation from this analysis is that the untrusted sub-cases had higher processing times compared to their trusted counterparts. This is because the untrusted sub-cases required querying the *Allow List* list indicies to determine if the measurements hash already existed. The process of exploring the entire *Allow List* consumes additional time. However, for the trusted case, the measurement hash is already there which in the worst case only need to search for the whole list.

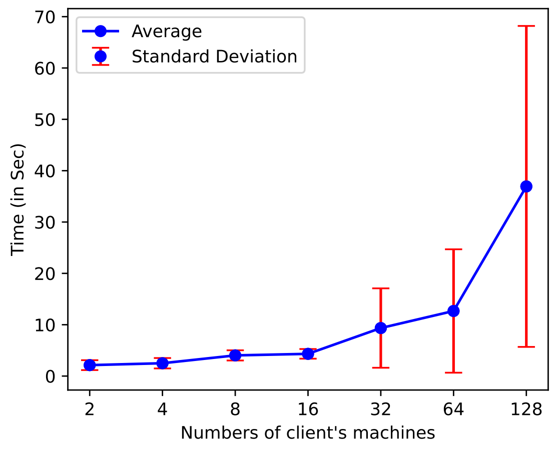


Fig. 8: Variation of evaluation time with an increasing number of client's machines

5.2 TPC Scalability Test

5.2.1 Environment Setup

To test the scalability of TPC, we deployed the project on AWS with the *t2.micro* VM with 1 core CPU, 1 GB RAM and 50 GB SSD storage. The configuration of *Verification Cluster* is exactly the same as given in Section 5.1.1. The number of VM is varied in an exponential form from 2 to 128. As the type of *TPM* does not change the performance of *Verification Cluster*, we keep it simple by using *Local TPM* for this test.

The *Trust Agent*, *Kernel Module* and *Load Generator* are installed with the necessary libraries on a test VM and a VM Image is constructed. The *Load Generator* is set to generate an average load. An AWS Launch Template is later defined with the previously generated Test Image. A series of steps are performed as follows: a) create the set of machines K and get the ID and token, b) create in parallel all the machine instances from the AWS Launch Template and get the IP address, update the ID, token and *Verification Cluster* address in the *Settings* of each machine, c) Start and run the test for the 30 minutes time period, d) Terminate the running machines, e) Delete the

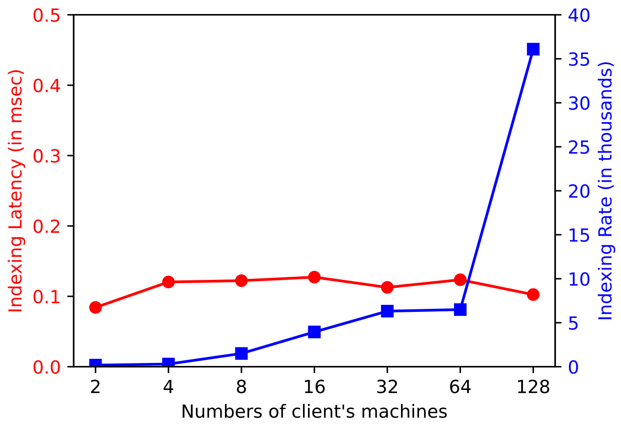


Fig. 9: Performance variation of OpenSearch with the varying number of client machines

machine IDs and f) Get the results. A Python script is written to automate these experiment steps and the results are evaluated.

5.2.2 TPC *Scalability Results*

Evaluation Time Variation. The results in Figure 8 show the total time consumption against the number of machines attached to the *Verification Cluster*. It is reasonable to observe an exponential growth pattern due to the exponential increase in the number of machines. As the number of machines increases exponentially, the total evaluation time exhibits a linear trend on a logarithmic scale, indicating that the evaluation time grows at a slower rate compared to the exponential increase in the number of machines.

It is worth noting that the observed evaluation time values remain within reasonable limits. The fact that the maximum time is 35.6 seconds for 128 VMs, while other configurations have evaluation times less than 20 seconds, suggests that the system can handle the workload effectively. These values clearly indicate that the system's performance is within an acceptable range considering the computational complexity and the number of machines involved.

VC's OpenSearch performance. Figure 9 shows the performance of OpenSearch in terms of indexing latency and indexing rate while varying the number of client machines connected. As the figure shows, the indexing latency remains comparable and around 0.1 msec across varying numbers of VMs. This indicates that the time it takes for OpenSearch to process and index data remains stable, regardless of the number of client machines involved. The consistent indexing latency implies the OpenSearch on *Verification Cluster* is able to effectively distribute and manage the indexing workload across the client machines. On the other hand, the indexing rate shows a noticeable variation with the increasing number of machines. As the indexing rate refers to the speed at which OpenSearch can process and index data, this observation aligns with the expectation that a larger number of machines allows for parallelisation and distributed processing, leading to higher indexing rates. As the number of client machines increases to 128, the indexing rate reaches around 36.2, which demonstrates a substantial boost in performance.

VC's RDS (Postgres) performance. In our evaluation of VC's RDS (Relational Database Service) as shown in Figure 10, we measured the throughput by varying the number of client

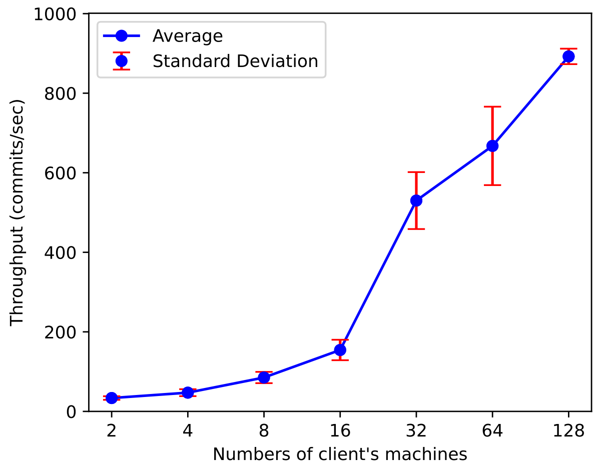


Fig. 10: Performance variation of RDS with the varying number of client machines

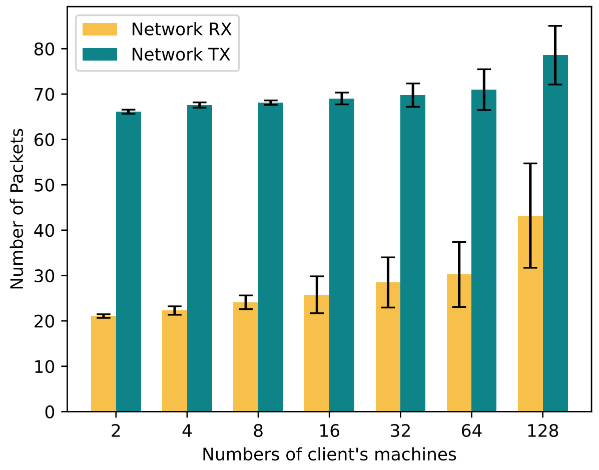
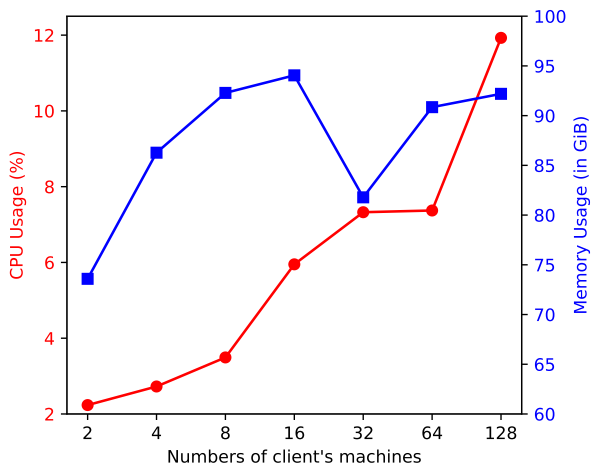


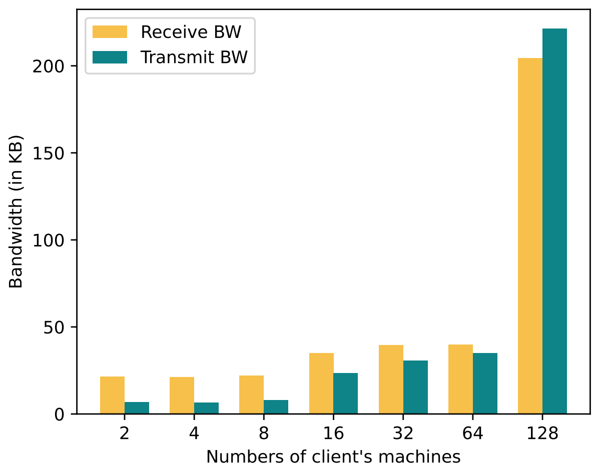
Fig. 11: Kafka performance variation

machines. We observed that the throughput consistently increases as the number of machines increases, ranging from 33.62 for 2 machines to 892.45 for 128 machines. In the case of RDS, the increase in throughput with the number of VMs indicates that the system benefits from parallel processing and the ability to distribute the workload across multiple nodes. As the number of VMs increases, RDS can handle more requests concurrently, resulting in higher throughput. The observed increase in throughput from 40 to 875 demonstrates the scalability and performance capabilities of RDS. This improvement is particularly significant, highlighting the advantages of leveraging additional VMs to achieve parallel processing and increase system throughput.

VC's Kafka performance. Next, we evaluated the performance of Kafka as shown in Figure 11. In this evaluation, we focused on measuring and plotting the Network receive (RX) and Network transmit (TX) metrics. The figure shows a slight increase in both values as the number of machines increases. Specifically, for network RX, the value varies from 21.07 packets for 2 machines to 43.22 packets for 128 machines, while for network TX, the value ranges from 66.12 packets for 2 machines to 78.57 packets for 128 machines. The slight increase in network RX packets signifies that VC's Kafka's ability to handle incoming data scales well as the number of machines

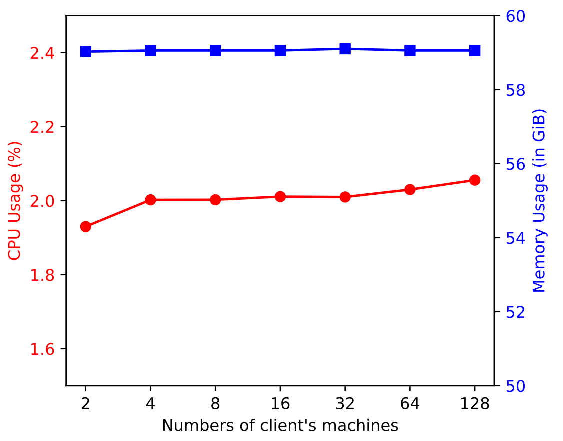


(a) CPU and memory usage for varying client machine

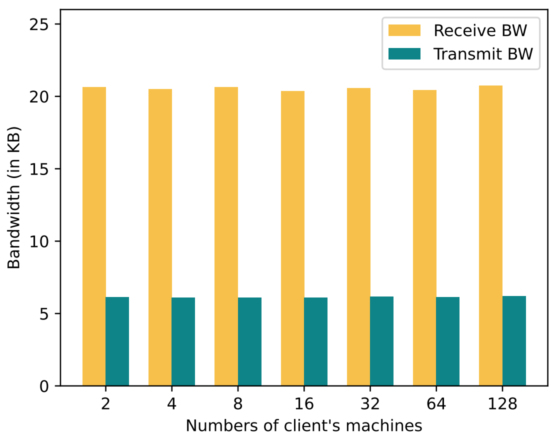


(b) Receive and transmit bandwidth for varying client machine

Fig. 12: Performance of AMP processors with the varying number of client machines



(a) CPU and memory usage for varying client machine



(b) Receive and transmit bandwidth for varying client machine

Fig. 13: Performance of MCP processors with the varying number of client machines

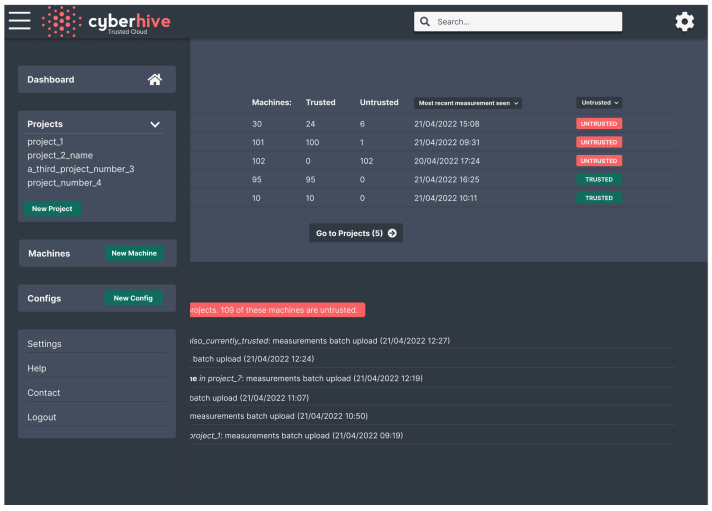
increases. The distributed nature of Kafka allows for parallel processing and efficient handling of incoming messages from multiple machines.

Similarly, the increase in network TX packets implies that as the number of machines increases, Kafka is transmitting a higher volume of network packets. The distributed architecture of Kafka enables efficient transmission of data to multiple machines consuming the data.

VC's AMP's performance. Figure 12 shows the performance of the AMP processor executing in the Kubernetes cluster. The observed CPU usage (see Figure 12a) in the processor executing as an AMP processor showcases the system's ability to efficiently utilize computational resources. Initially, the CPU usage increases from 5% for 2 machines to 11% for 16 machines, indicating that as more VMs are added, the system effectively distributes and utilizes processing power. The subsequent decrease in CPU usage to 7% for 32 machines may be attributed to optimization techniques or workload balancing mechanisms employed within the cluster. Continuing from 32 machines, the gradual increase in CPU usage to 10.5% for 128 machines demonstrates the system's ability to scale and efficiently handle increased computational demands. This upward trend signifies the successful utilization of CPU resources as the number of attached machines grows. It's important to emphasize that the overall CPU usage remains relatively low, indicating that the AMP processor is effectively managing resources and operating within an optimal range. Similarly, the gradual increase in memory usage as the number of machines increases reflects the system's ability to accommodate growing data requirements. From 2 GiB for 2 VMs to 12 GiB for 128 VMs, the smooth and consistent rise in memory usage highlights the system's scalability and ability to efficiently store and process larger volumes of data.

Figure 12b shows the receive and transmit bandwidth of the same AMP processor. As the result shows, there is a slight increase in receive and transmit bandwidth from 2 VMs to 64 VMs which demonstrates the system's capability to handle increased data transfer demands. This growth showcases the ability of the processor executing as a Kubernetes cluster to efficiently receive and transmit data across the network. The sudden increase in values from (44, 42) KB to (200, 220) KB respectively is due to the system's optimized handling of network communication provided by Kubernetes.

VC's MCP's performance. The performance of the Measure-



Machines:	Trusted	Untrusted	Most recent measurement seen	Untrusted
30	24	6	21/04/2022 15:08	UNTRUSTED
101	100	1	21/04/2022 09:31	UNTRUSTED
102	0	102	20/04/2022 17:24	UNTRUSTED
95	95	0	21/04/2022 16:25	TRUSTED
10	10	0	21/04/2022 10:11	TRUSTED

rojects. 109 of these machines are untrusted.

```
also_currently_trusted: measurements batch upload (21/04/2022 12:27)
 batch upload (21/04/2022 12:24)
ne in project_7: measurements batch upload (21/04/2022 12:19)
batch upload (21/04/2022 11:07)
measurements batch upload (21/04/2022 10:50)
roject_1: measurements batch upload (21/04/2022 09:19)
```

Fig. 14: User Interface of TPC

ment Configuration Processor (MCP) executing as a Kubernetes cluster is shown in Figure 13. As shown in Figure 13a, the CPU usage is consistent and comparable varying slightly from 1.9% for 2 machines to 2.1% for 128 machines. This stability in CPU usage suggests that the MCP cluster maintains a balanced and optimized allocation of resources, ensuring smooth operation even as the number of machines increases. The figure also shows the consistent memory usage which showcases the cluster's ability to handle the data requirements effectively, contributing to its overall performance and stability.

Figure 13b shows the receive and transmit bandwidth which is unchanged with 20 KB and 5.4 KB respectively. The consistent bandwidth values demonstrate the system's ability to sustain a reliable and consistent network throughput, regardless of the number of VMs involved.

5.2.3 TPC *Graphical User Interface*

The user interface of the TPC framework effectively balances robust security features with user accessibility. This ensures that administrators and IT managers can securely manage and monitor the machines and projects regardless of the applications executing on the system. The interface is designed to be intuitive, allowing users to define and associate projects and machines, and configure *Include lists*, *Exclude lists*, and *Allow lists* easily as shown in Figure 14. At the same time, the interface supports real-time monitoring, enabling users to observe the live performance and security status of machines. This allows IT personnel to manage security for the zero-trust without compromising usability, making it a powerful tool for maintaining both operational efficiency and rigorous security standards.

6 Discussion

Experiment results highlight TPC's performance in terms of client machine tracking, system scalability and flexibility. While this encourages the use of TPC for tracking the trust state, there are a few other aspects to be considered as given below.

Making a generic monitoring framework. The *Kernel Module* component of TPC relies on the system's kernel version to capture the extended IMA-based measurements. Although the *Kernel Module* supports all the currently available kernel versions, an update is required to use *Kernel Module* on a new kernel version. We are working on making the *Kernel Module* generic enough to be used on any kernel version. [32] uses an extended Berkely Packet Filters (eBPF)-based system to monitor the basic kernel operations. Developing similar techniques to adapt TPC to capture the extended-IMA measurements is also part of our future work plan.

Automate the decision-making process. The *Verification Cluster* notifies the administrator of the client machine's becoming untrusted. A follow-up operation is performed by the administrator to verify the reason for making the client machine untrusted. A system re-evaluation is performed after this process either making the system state permanently untrusted or changing the system state to trusted. This manual process is time-consuming and relies on the administrator's skills. An intuitive model is required to automate the decision-making process. One of our future work is to leverage a machine learning model and train it with the previous system measurements which can later detect a system state change and makes the decision.

Cascading of security providers. As the result shows, using SGX-SWTPM is slower but is the most secure method. We are planning to work on a cascading security provider to use SGX-SWTPM periodically, according to administrator-defined parameters while utilising either SWTPM or a local security provider for the remaining measurements. This can guarantee the same level of security as the SGX-SWTPM while consuming minimal time.

Reuse the Verification Knowledge. The boundary of TPC's verification process is bounded on a per-project basis i.e., each client machine has an associated set of lists (*Allow List*, *Include List*) in the *Verification Cluster* attached for the verification process. Since TPC is based on a zero-trust framework, any update in the client's machine needs to be verified keeping the system trusted. However, there are some generic updates e.g., an update in Chrome which can happen in multiple machines. The knowledge of successful verification on one machine can be utilised for another machine as well. We are currently working on automatically producing *Allow List*s for popular applications which are permitted to be on the machine.

Bring trust to the verification process. The administrator is considered to be a trusted entity that verifies the machine status. They are involved in updating the *Allow List* if the change is found to be trusted. However, the update can be overwritten and may lead to a lack of trust. To guarantee a chain of trust for the administrator and to make the updates immutable, we are planning to use Git hash values or Distributed Ledger Technology in future work. This will let the client know of any changes. In addition to this, any change in the *Allow List* can easily be reused for different sets of machines.

Attack evaluation in staging environment. We are working on testing the TPC framework to assess its effectiveness against various security threats, such as Data Manipulation and Eavesdropping attacks. We plan to simulate these attacks within our staging cluster environment, allowing us to rigorously evaluate how well TPC mitigates these threats. This testing will help us understand the TPC framework's robustness in a live setting and guide further refinements to enhance its security capabilities for the production-ready environment.

Combining TPC with existing Network-based zero-trust framework. As discussed, TPC offers protection for the host system-level threats. One of our future work is to integrate the TPC framework with existing network-based zero-trust frameworks. While the network-based zero-trust framework secures the perimeter and manages access controls, TPC ensures that the system itself remains "Trusted". The result is a unified Zero Trust architecture that offers enhanced protection against a broader range of threats from network-based attacks to internal system vulnerabilities. We would also like to evaluate the whole framework

7 Conclusion

In this paper, we proposed and evaluated TPC in different environments. We evaluated the TPC for different security providers with the *Verification Cluster* in a staging environment. The result shows that TPC is able to detect the system's untrusted state in less than a minute as compared to 211 days for a middle-sized organisation. The evaluation also shows that the *Verification Cluster*, at the core of the system, is scalable with each component showing comparable performance with the increasing number of client machines.

Acknowledgment

This research was supported by the Knowledge Transfer Partnership (KTP) project, *Trusted Public Cloud* (id 11289).

References

[1] S. Sha, "The reliability of enterprise applications," *Communications of the ACM*, vol. 63, no. 1, pp. 38–45, 2019.

[2] D. Wermke, N. Huaman, C. Stransky, N. Busch, Y. Acar, and S. Fahl, "Cloudy with a chance of misconceptions: Exploring users' perceptions and expectations of security and privacy in cloud office suites," in *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, 2020, pp. 359–377.

[3] P. Sun, "Security and privacy protection in cloud computing: Discussions and challenges," *Journal of Network and Computer Applications*, vol. 160, p. 102642, 2020.

[4] F. Wang, B. Diao, T. Sun, and Y. Xu, "Data security and privacy challenges of computing offloading in fins," *IEEE Network*, vol. 34, no. 2, pp. 14–20, 2020.

[5] J. A. De Guzman, K. Thilakarathna, and A. Seneviratne, "Security and privacy approaches in mixed reality: A literature survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–37, 2019.

[6] D. N. Jha, G. Lenton, J. Asker, D. Blundell, and D. Wallom, "Trusted platform module based privacy in public cloud: Challenges and future perspective," *IEEE IT Professional*, pp. 1–5, 2022.

[7] M. Ma, Z. Yin, S. Zhang, S. Wang, C. Zheng, X. Jiang, H. Hu, C. Luo, Y. Li, N. Qiu *et al.*, "Diagnosing root causes of intermittent slow queries in cloud databases," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1176–1189, 2020.

[8] H. Tabrizchi and M. K. Rafsanjani, "A survey on security challenges in cloud computing: issues, threats, and solutions," *The journal of supercomputing*, vol. 76, no. 12, pp. 9493–9532, 2020.

[9] R. Khandelwal, T. Linden, H. Harkous, and K. Fawaz, "Prisec: A privacy settings enforcement controller," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[10] C. Stergiou, K. E. Psannis, B. B. Gupta, and Y. Ishibashi, "Security, privacy & efficiency of sustainable cloud computing for big data & iot," *Sustainable Computing: Informatics and Systems*, vol. 19, pp. 174–184, 2018.

[11] L. Zhao and D. Lie, "Is hardware more secure than software?" *IEEE Security & Privacy*, vol. 18, no. 5, pp. 8–17, 2020.

[12] S. F. J. J. Ankergård, E. Dushku, and N. Dragoni, "State-of-the-art software-based remote attestation: Opportunities and open issues for internet of things," *Sensors*, vol. 21, no. 5, p. 1598, 2021.

[13] S. Ameer, L. Praharaj, R. Sandhu, S. Bhatt, and M. Gupta, "Zta-iot: A novel architecture for zero-trust in iot systems and an ensuing usage control model," *ACM Transactions on Privacy and Security*, 2024.

[14] Y. Liu, X. Xing, Z. Tong, X. Lin, J. Chen, Z. Guan, Q. Wu, and W. Susilo, "Secure and scalable cross-domain data sharing in zero-trust cloud-edge-end environment based on sharding blockchain," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[15] B. Ali, M. A. Gregory, S. Li, and O. A. Dib, "Implementing zero trust security with dual fuzzy methodology for trust-aware authentication and task offloading in multi-access edge computing," *Computer Networks*, vol. 241, p. 110197, 2024.

[16] Q. Shen and Y. Shen, "Endpoint security reinforcement via integrated zero-trust systems: A collaborative approach," *Computers & Security*, vol. 136, p. 103537, 2024.

[17] R. Geng, C. Fang, S. Guo, D. Kang, B. Lyu, S. Zhu, and P. Cheng, "Flowpinpoint: Localizing anomalies in cloud-client services for cloud providers," *IEEE Transactions on Cloud Computing*, 2023.

[18] B. Shen, H. Yu, P. Hu, H. Cai, J. Guo, B. Xu, and L. Jiang, "A cloud-edge collaboration framework for generating process digital twin," *IEEE Transactions on Cloud Computing*, 2024.

[19] R. Xin, P. Chen, P. Grosso, and Z. Zhao, "A fine-grained robust performance diagnosis framework for run-time cloud applications," *Future Generation Computer Systems*, 2024.

[20] A. Noor, D. N. Jha, K. Mitra, P. P. Jayaraman, A. Souza, R. Ranjan, and S. Dustdar, "A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments," in *2019 IEEE 12th international conference on cloud computing (CLOUD)*. IEEE, 2019, pp. 156–163.

[21] X. Zhou, B. Ahmed, J. H. Aylor, P. Asare, and H. Alemzadeh, "Hybrid knowledge and data driven synthesis of runtime monitors for cyber-physical systems," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[22] N. Rajesh, H. Devarajan, J. C. Garcia, K. Bateman, L. Logan, J. Ye, A. Kougkas, and X.-H. Sun, "Apollo: An ml-assisted real-time storage resource observer," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 147–159.

[23] B. Yang, W. Xue, T. Zhang, S. Liu, X. Ma, X. Wang, and W. Liu, "End-to-end i/o monitoring on leading supercomputers," *ACM Transactions on Storage*, vol. 19, no. 1, pp. 1–35, 2023.

[24] U. Demirbaga, A. Noor, Z. Wen, P. James, K. Mitra, and R. Ranjan, "Smartmonit: Real-time big data monitoring system," in *2019 38th symposium on reliable distributed systems (SRDS)*. IEEE, 2019, pp. 357–3572.

[25] W. Huang, T. Li, J. Liu, P. Xie, S. Du, and F. Teng, "An overview of air quality analysis by big data techniques: Monitoring, forecasting, and traceability," *Information Fusion*, vol. 75, pp. 28–40, 2021.

[26] "Amazon cloudwatch," https://aws.amazon.com/cloudwatch/, accessed: 2024-03-07.

[27] "Azure monitor," https://azure.microsoft.com/en-gb/products/monitor/, accessed: 2024-03-07.

[28] "Dynatrace: Cloud monitoring," https://www.dynatrace.com/platform/applications-microservices-monitoring/, accessed: 2024-03-07.

[29] K. Vijayakumar and C. Arun, "Continuous security assessment of cloud based applications using distributed hashing algorithm in sdlc," *Cluster Computing*, vol. 22, no. 5, pp. 10 789–10 800, 2019.

[30] T. Zhang, M. L. Rahman, H. M. Kamali, K. Z. Azar, and F. Farahmandi, "Sipguard: Run-time system-in-package security monitoring via power noise variation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

[31] "Datadog: Modern monitoring & security," https://www.datadoghq.com/, accessed: 2024-03-07.

[32] D. N. Jha, G. Lenton, J. Asker, D. Blundell, and D. Wallom, "Holistic runtime performance and security-aware monitoring in public cloud environment," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 1052–1059.

[33] S. Laurén and V. Leppänen, "Virtual machine introspection based cloud monitoring platform," in *Proceedings of the 19th International Conference on Computer Systems and Technologies*, 2018, pp. 104–109.

[34] S. Geissler, S. Lange, F. Wamser, T. Zinner, and T. Hoßfeld, "Komon—kernel-based online monitoring of vnf packet processing times," in *2019 International Conference on Networked Systems (NetSys)*. IEEE, 2019, pp. 1–8.

[35] A. Mayer, P. Loreti, L. Bracciale, P. Lungaroni, S. Salsano, and C. Filsfils, "Performance monitoring with hˆ2: Hybrid kernel/ebpf data plane for srv6 based hybrid sdn," *Computer Networks*, vol. 185, p. 107705, 2021.

[36] M. H. Nasir, S. A. Khan, M. M. Khan, and M. Fatima, "Swarm intelligence inspired intrusion detection systems—a systematic literature review," *Computer Networks*, p. 108708, 2022.

[37] Z. Halim, M. N. Yousaf, M. Waqas, M. Sulaiman, G. Abbas, M. Hussain, I. Ahmad, and M. Hanif, "An effective genetic algorithm-based feature selection method for intrusion detection systems," *Computers & Security*, vol. 110, p. 102448, 2021.
[38] S. Agrawal, S. Sarkar, O. Aouedi, G. Yenduri, K. Piamrat, M. Alazab, S. Bhattacharya, P. K. R. Maddikunta, and T. R. Gadekallu, "Federated learning for intrusion detection system: Concepts, challenges and future directions," *Computer Communications*, 2022.
[39] H. Yan, X. Li, W. Zhang, R. Wang, H. Li, X. Zhao, F. Li, and X. Lin, "Automatic evasion of machine learning-based network intrusion detection systems," *IEEE Transactions on Dependable and Secure Computing*, 2023.
[40] H. Sun, M. Chen, J. Weng, Z. Liu, and G. Geng, "Anomaly detection for in-vehicle network using cnn-lstm with attention mechanism," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 10, pp. 10 880–10 893, 2021.
[41] A. Oseni, N. Moustafa, G. Creech, N. Sohrabi, A. Strelzoff, Z. Tari, and I. Linkov, "An explainable deep learning framework for resilient intrusion detection in iot-enabled transportation networks," *IEEE Transactions on Intelligent Transportation Systems*, 2022.
[42] Y. Wu, Y. Xie, X. Liao, P. Zhou, D. Feng, L. Wu, X. Li, A. Wildani, and D. Long, "Paradise: real-time, generalized, and distributed provenance-based intrusion detection," *IEEE Transactions on Dependable and Secure Computing*, 2022.
[43] J. Cui, H. Sun, H. Zhong, J. Zhang, L. Wei, I. Bolodurina, and D. He, "Collaborative intrusion detection system for sdvn: A fairness federated deep learning approach," *IEEE Transactions on Parallel and Distributed Systems*, 2023.
[44] M. Verkerken, L. D'hooge, D. Sudyana, Y.-D. Lin, T. Wauters, B. Volckaert, and F. De Turck, "A novel multi-stage approach for hierarchical intrusion detection," *IEEE Transactions on Network and Service Management*, 2023.
[45] M. Gorbett, H. Shirazi, and I. Ray, "Local intrinsic dimensionality of iot networks for unsupervised intrusion detection," in *Data and Applications Security and Privacy XXXVI: 36th Annual IFIP WG 11.3 Conference, DBSec 2022, Newark, NJ, USA, July 18–20, 2022, Proceedings*. Springer, 2022, pp. 143–161.
[46] I. Martins, J. S. Resende, P. R. Sousa, S. Silva, L. Antunes, and J. Gama, "Host-based ids: A review and open issues of an anomaly detection system in iot," *Future Generation Computer Systems*, 2022.
[47] M. Higgins, D. Jha, and D. Wallom, "Spatial-temporal anomaly detection for sensor attacks in autonomous vehicles," in *2023 IEEE Smart World Congress (SWC)*. IEEE, 2023, pp. 783–788.
[48] O. Ghazali, A. M. Tom, H. M. Tahir, S. Hassan, S. A. Nor, and A. H. Mohd, "Security measurement as a trust in cloud computing service selection and monitoring," *Journal of Advances in Information Technology Vol*, vol. 8, no. 2, 2017.
[49] L. Guo, H. Yang, K. Luan, Y. Luo, L. Sun *et al.*, "A trust model based on characteristic factors and slas for cloud environments," *IEEE Transactions on Network and Service Management*, 2023.
[50] S. Rizvi, K. Karpinski, B. Kelly, and T. Walker, "Utilizing third party auditing to manage trust in the cloud," *Procedia Computer Science*, vol. 61, pp. 191–197, 2015.
[51] G. Aghaee Ghazvini, M. Mohsenzadeh, R. Nasiri, and A. Masoud Rahmani, "Mmlt: A mutual multilevel trust framework based on trusted third parties in multicloud environments," *Software: Practice and Experience*, vol. 50, no. 7, pp. 1203–1227, 2020.
[52] A. Balcao-Filho, N. Ruiz, F. Rosa, R. Bonacin, and M. Jino, "Applying a consumer-centric framework for trust assessment of cloud computing service providers," *IEEE Transactions on Services Computing*, 2021.
[53] D. Wallom, A. Ruan, and D. Blundell, "Porridge: A method of providing resilient and scalable cloud-attestation-as-a-service," in *12th International Conference on System Safety and Cyber-Security 2017 (SCSS)*. IET, 2017, pp. 1–6.
[54] A. Muñoz and E. B. Fernandez, "Tpm, a pattern for an architecture for trusted computing," in *Proceedings of the European Conference on Pattern Languages of Programs 2020*, 2020, pp. 1–8.
[55] C. Han, T. Kim, W. Lee, and Y. Shin, "S-zac: Hardening access control of service mesh using intel sgx for zero trust in cloud," *Electronics*, vol. 13, no. 16, p. 3213, 2024.

Devki Nandan Jha is currently a Lecturer at Newcastle University, Newcastle Upon Tyne, UK. He is also a visiting researcher at the Oxford e-Research Centre, University of Oxford. Previously, he was a Research Associate with Oxford e-Research Centre, University of Oxford, Oxford and CyberHive Ltd., Newbury, UK. He has a PhD in Computer Science from Newcastle University, Newcastle Upon Tyne, UK. His research interests include cloud computing, internet of things, trust and security, and machine learning.

Graham Lenton is currently the Head of Engineering at CyberHive Ltd. Before joining CyberHive, Graham led the development team at BBC Monitoring, UK. He has more than 25 years of Industry experience in development and management. His research interests include cloud computing, trusted computing, secure networking and software integration.

James Asker is a Development Supervisor at CyberHive, and has 15 years of Industry experience in systems engineering and administration. His research interests are secure cloud computing and Linux kernel security.

David Blundell is the founder, MD and CTO of CyberHive Ltd. His current research interests are secure distributed computing, autonomous vehicles and post-quantum encryption.

Martin Higgins (S'19) received a BSc in Physics from Queen Mary, University of London, in 2011, an MSc from Imperial College London, UK, in 2012, MRES from the University of Strathclyde in 2018 and PhD from Imperial College in 2022. He is currently a research associate at the University of Oxford on the Digital Security by Design Project. His research interests lie in power systems, cyber-security, false data injection attacks, and autonomous vehicles.

David C.H. Wallom is currently an Associate Director of Innovation with the Oxford e-Research Centre, University of Oxford. He was the Technical Director of the U.K. National Grid Service until the closure of the service and is the Current Chair of the European Grid Infrastructure Federated Cloud Task Force. His current research interests include applications and reuse of e-infrastructure, as well as the application of high-performance computing techniques and cybersecurity.