APPARENT: AI-Powered Platform Anomaly Detection in Edge Computing

Chandrajit Pal, Sangeet Saha, Xiaojun Zhai, Gareth Howells, and Klaus D. McDonald-Maier

Abstract—Embedded systems serving as IoT nodes are often vulnerable to malicious and unknown runtime software that could compromise the system, steal sensitive data, and cause undesirable system behaviour. Commercially available embedded systems used in automation, medical equipment, and automotive industries, are especially exposed to this vulnerability since they lack the resources to incorporate conventional safety features and are challenging to mitigate through conventional approaches. We propose a novel system design coined as APPARENT which can identify program characteristics by monitoring and counting the maximum possible low-level hardware events from Hardware Performance Counters (HPCs) that occur during the program's execution and analyse the correlation among the counts of various monitored events. To further utilise these captured events as features we propose a self-supervised machine learning algorithm that combines a Graph Attention Network GAT and a Generative Topographic Mapping GTM to detect unusual program behaviour as anomalies to enhance the system security. Our proposed methodology takes advantage of attributes like program counter, cycles per instruction, and physical and virtual timers at various exception levels of the embedded processor to identify abnormal activity. APPARENT identifies unknown program behaviours not present in the training phase with an accuracy of over 98.46% on Autobench EEMBC benchmarks.

Index Terms—Generative Topographic Mapping (GTM), Graph Attention Network (GAT), Cycles Per Instruction (CPI), Control Flow Graph (CFG), Gated Recurrent Unit (GRU), Machine Learning (ML), Anomaly Detection.

1 Introduction

Currently, Internet of Things (IoT) technology deployment is expanding across all application domains. However, there are concerns over security issues related to the quick and widespread adoption of such connected devices. IoT devices have limited processing power and storage capacity, which makes it difficult to apply security patches. Hence, securing resource-constrained IoT devices and the systems they communicate with is now of paramount importance [1]. Current research mainly focuses on attacks undermining the security of data exchange and communication between IoT devices and other systems. Nevertheless, the existing threat patterns indicate that in addition to attacks on IoT networks, embedded systems IoT devices are now becoming vulnerable to attack points within an IoT botnet [2], following commands from the adversary since these nodes make decisions as front-end devices (Fig. 1). This necessitates the attack detection and monitoring system to be installed as close as possible to the traffic data source, mainly at the embedded IoT platforms, thereby improving service latency and conserving limited bandwidth by eliminating costly server communication.

As we move down the hierarchy from cloud servers to embedded IoT platforms (Fig. 1), computing units gradually get more constrained in terms of power and computing resources, with minimal resources in IoT devices. Hence, *anomaly detection, using intelligent techniques in a resource-constrained embedded system in real-time while maintaining the performance is a challenging problem in edge computing*. The techniques of developing anomaly detection models that work with the diverse embedded architectures found in IoT platforms are thus undoubtedly challenging. Generic security measures through the use of cryptographic algorithms and security protocols are often incompatible with specific embedded architectures which are typically specialised to perform a particular function repetitively [3]. They are also constrained by limited resources use custom firmware and often even run without an operating system, making it challenging to implement security mechanisms with traditional antivirus programs which look for deviations in regular program execution.

There exist state-of-the-art software and hardware-based embedded device security solutions [4], [5]. This includes Hardware intrinsic security [6] or Physical Unclonable Function (PUF) [4]. Additionally, some work [7], and [8] concentrate on identifying software failure, tampering, and malicious codes in embedded architectures. The primary drawback of these methodologies is that they necessitate keeping private and sensitive details in the platform as "valid" templates like the control-flow-graph CFG, maintaining it consumes more storage requires a considerable amount of processing power and is often exposed to attack. Hence, these software-based solutions are unsuited for resource-constrained IoT devices, which typically have limited processing power, memory, and storage capacity. Another primary drawback of software-based security solutions is their intrusive nature and dependency on the existing software stack of a system, which is problematic in many cases. Malicious codes dynamically and continuously change their internal structure and attack patterns using obfuscation techniques [9]. Thus, static software-based security

• *All the authors are with the School of Computer Science and Electronics Engineering, University of Essex, Colchester, UK, CO43SQ.*

Corresponding author: Xiaojun Zhai

Manuscript received xxxx; revised xxxx.

solutions become less effective in such cases due to their dependence on internal software modelling. This limitation calls for exploring alternative approaches to ensure the security of IoT devices.

Hardware Performance Counters (HPCs) offer a promising avenue for anomaly detection in resource-constrained environments. HPCs are specialised hardware units that monitor and record low-level hardware events during program execution at runtime aiding in anomaly detection. These features of hardware events from an embedded device are difficult to compromise and bypass by obfuscation techniques. These events can provide valuable insights into program behaviour and can be used to detect deviations from the expected execution patterns, even in the absence of complex software-based analysis. Recently, the authors in [10], [11] proposed HPC-based anomaly detection.

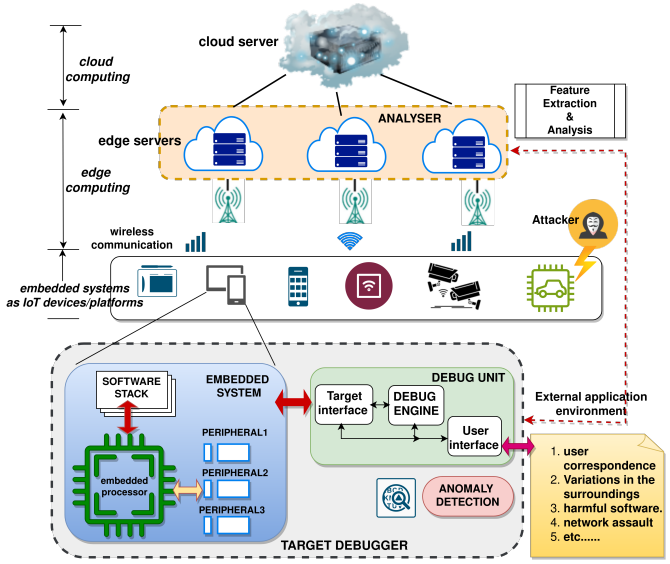


Fig. 1. Illustrating an embedded system as an IoT device exposed to an attack

A time series of HPC measurements (special register values) provides a temporal profile of the code under execution which acts as an important indicator of system behaviour. The HPC time series on a *known-good* embedded device describes the predicted temporal features of programs running on the embedded processor. These special-purpose counters are used to precisely and accurately record hardware events in real time. When a system functions normally, i.e., when no errors are found in the system—it displays a profile; any variations from this profile point to a potential anomaly in the system. Moreover, these low-level features are robust and hard to compromise. However, the exploitation of HPCs for anomaly detection is still in its infancy, as it is not straightforward to collect and analyse HPCs for anomaly detection without operating system support, as it is primarily used for performance tuning.

A few studies [10], [12], [13], [14], [15] have developed strategies to detect anomalies using machine learning classification of HPC measurements. Some studies demonstrated their approach with supervised learning approaches and often considered single HPC event series to reduce profiling overhead during classification. Authors in [16] employed five supervised machine learning algorithms trained offline only on HPC events which may be a good choice for high-performance systems. Wang et al. [17] detected side-channel attacks using low-level HPC events leveraging supervised multimodal machine learning algorithms like regression, neural networks, decision trees, and rule-based algorithms capable of handling both linear and nonlinear prediction problems. Nevertheless, deciding how to best integrate information from multiple modalities is crucial. There are various fusion techniques, but the process of choosing the right one can significantly impact performance. Again, supervised learning techniques computed from a large number of HPC events [18] may significantly increase the system overhead for resource-constrained devices. A wider range of data points often leads to confusion, while detailing and missing the bigger performance bottlenecks. Focusing on the most relevant counters is crucial. Moreover, offline trained supervised classifiers [19] deployed in security systems make it difficult to learn from new data and adapt to emerging threats continuously. There are also some inherent problems regarding the availability of labelled datasets for supervised learning models. They are difficult and time-consuming to acquire, especially for diverse embedded systems with varying behaviours and potential anomalies. Even if training data is available, the model should be trained on sufficient and representative samples from various operating environments. However, gathering such extensive datasets takes time, especially for systems with changing environmental conditions. This challenge motivates the exploration of alternative approaches that can effectively detect anomalies without relying on extensive labelled training data, making it more suitable for dynamic embedded environments where software and hardware configurations may change.

Some papers like [20] and [21] leveraged unsupervised learning on microarchitectural features from HPCs and network traffic data to detect anomalies arising out of threats. The commonality in their studies lies in finding a common structure between the datasets meant for one class classification to perform either clustering or dimensionality reduction. For faster adaptation to unknown datasets, authors in [22] leveraged unsupervised transfer learning where they transferred knowledge from the pre-trained model and thereby learned faster with less data than purely unsupervised learning. However, in scenarios where labelled data is scarce and performance is a top priority, ensemble learning might be a better option. Sayadi [23] utilised diverse models and potentially achieved higher accuracy by leveraging ensemble learning techniques to consider the effect of lowering the number of HPC features on malware detector performance, thereby improving the performance of hardware-assisted malware detectors. These solutions work well on platforms that have sufficient computational resources but might not be a good design choice for resource-constrained edge platforms. Therefore, to partially reduce the dependency upon annotated/labelled data it is important to learn continually the underlying characteristics of unlabelled data and dependency among the data events. *Hence, for more accurate classification, it is necessary to collect multiple HPC events and find a correlation among them by continually learning the newly encountered program data with self-supervision while*

maintaining reasonable overhead.

In this paper, we propose APPARENT, which is independent of high-level software environments such as operating systems (OS) and source programs. Thus, it can be applied to a broader range of embedded systems to extract, using an on-chip debugger, and analyse low-level hardware information from the HPCs. Upon gathering the information, we applied self-supervised GTM [24] and GAT [25] classifiers to generate models that identify a program's expected execution trajectory using HPCs. If the predicted model output and the model's estimated output diverge from a realistic model of the program's expected trajectory, this indicates an abnormality in the system. Figure 1 denotes the overview of the APPARENT. The two important components consist of an on-chip debugger employed at the device level for collecting trace data. Once collected, the data are sent to the edger server (acting as an analyser) for analysis. Thanks to low-latency communication strategies [26], these data transmissions can be achieved with low timing overheads. The following summarises our primary contributions to this manuscript:

- APPARENT leverages low-level hardware events like program counter values, cycles per instruction, capability registers and timer values and uses them as distinct features for accurate program recognition.
- APPARENT employs self-supervised Generative Topographic Mapping (GTM) in combination with a Graph Attention Network (GAT) for classifying the underlying extracted hardware features in a standalone bare-metal environment to detect anomalies through similarity measurements of program counter values in conjunction with a multivariate time series analysis of multiple HPCs on various benchmarks. The self-supervised nature of GTM and GAT allows the system to continuously learn and adapt to new data, making it effective for unknown programs.
- APPARENT analyses multiple HPCs as a multivariable time series, capturing the complex relationships among different events. Any deviation of pattern from these learned relationships flags the presence of probable threats.
- APPARENT can accurately identify unknown program behaviours not present in the training phase with 98.46% accuracy on Autobench EEMBC benchmarks.

2 Related Work

A summary of earlier studies on embedded systems security as IoT nodes is provided in this section. The need for digital privacy in protecting personal data has increased due to the digitisation of information for quick access, as mentioned in Section I [27], [28]. The authors in [29] systems show to leverage of on-chip debug information to detect abnormalities in embedded system program execution, while other research has examined the behaviour and frequency of code modifications meant to harm a system or its user. Secure program execution has been addressed by Batina et al. [30] by focusing on the particular problem of how to make sure a program follows its intended behaviour. To establish the basis for regulating acceptable program behaviour, they retrieved attributes from an embedded program. Software watermarks incorporated into protected software, serve as a distinctive identifier and prevent IP privacy. Haoyu and Iwendi [31], [32] studied the software watermarks after realizing the significant economic impact of software piracy.

To supplement traditional antivirus software, Mahdavifar et al. [33] created a malware detection method that automatically contrasts behaviour models with unidentified programs' runtime behaviour. Hao et al. [34] used a Control Flow Graph (CFG), much like in [30], to discover behavioural differences between malware and legitimate systems to identify breaches in protected embedded systems. Their system recognised sequences of system calls produced by the appropriate software and linked each performing process to a finite state machine (FSM). To improve system-wide efficiency without altering the architecture of the instruction set, Haq et al. [35] suggested various techniques for concealing information in compiled program binaries in their survey. Boufounos and Rane [36] developed an embedding system for secure search for the closest neighbour, including procedures for grouping and verifying applications. They also proved secure determination of signal similarity using machine learning and signal processing methodologies. As previously stated, software watermarks are unique features built into a program and act as identifiers [29]. It has been highlighted in [29], [37] that there are disadvantages to leveraging system calls for program identification, especially in embedded systems without a conventional operating system. These limitations include programs with few or no system calls, including those only arithmetic-based, and programs that lack distinctive behaviours during system calls that might not display a watermark.

Kadiyala et al. [38] investigated anomaly detection in a multicore environment and computed a boundary to distinguish between known benign and unknown malware code, however with programs of very short execution times. In another paper [16] they focused specifically on malware detection in systems with an operating system, by extracting HPCs at the system call level and applying supervised machine learning for classification. Authors in [39] utilise HPC values in a bare-metal environment but focus on using Principal Component Analysis to select micro-architectural events and then evaluate different supervised machine learning models. Authors in [40] and [41] proposed an unsupervised approach by leveraging a combination of GTM and Graph Theory (GT) strategy where GTM highlights system features, reducing variable dimensionality followed by computing similarity among samples. Subsequently, GT clusters them using networks, discriminating normal and anomalous entries. Similar graph theory concepts have been utilised where authors [42] proposed a multimodal spatial-temporal graph attention network (MST-GAT) which employs a multimodal graph attention network (M-GAT) and a temporal convolution network to capture the spatial-temporal correlation in multimodal time series in locating the most anomalous univariate time series variable. Authors in [43] focus on identifying and analyzing the challenges and vulnerabilities of using HPCs for security, including non-determinism, overcounting, and adversarial manipu-

lation. They seem to provide a broader overview of the challenges and pitfalls of using HPCs for various security applications, without restricting to only malware detection.

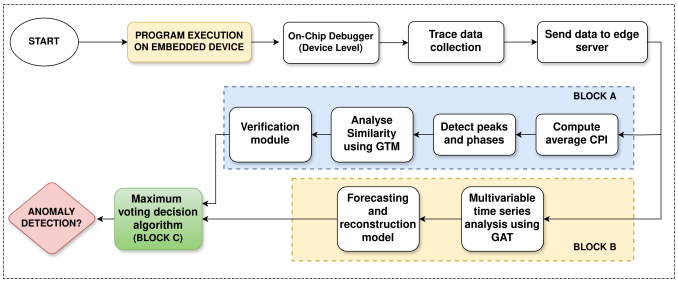


Fig. 2. Illustrating the flowchart of APPARENT

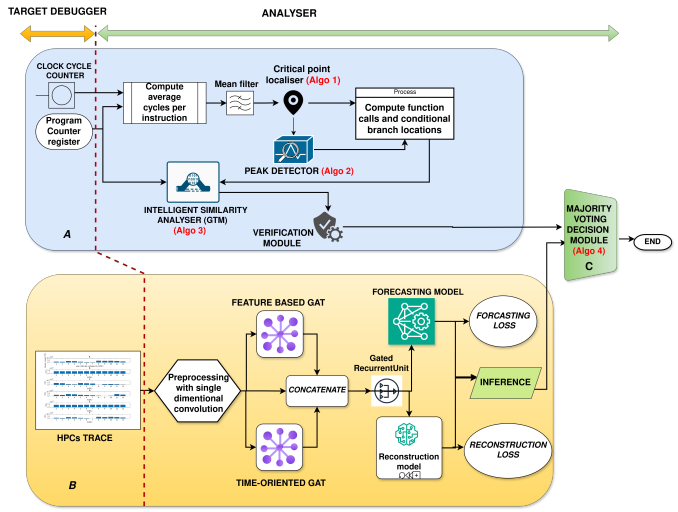


Fig. 3. Block A illustrates the procedure using GTM and Block B uses GAT. Block C computes the majority decision of an anomalous event after individually receiving decisions from both modules A and B.

APPARENT over state-of-the-art: Since most of the embedded hardware used as IoT nodes run without sophisticated OS [44], we plan to trace the low-level hardware events from multiple HPCs like program counters, clock cycle counters, physical and virtual timers at various exception levels unlike [11], [13]. Besides HPC events we have separately extracted the PC register values for computing the average cycles per instruction and apply some preprocessing algorithms to find the conditional branches and functional jumps to aid in anomaly detection. APPARENT can identify program characteristics by monitoring and counting the maximum possible low-level hardware events that occur during the program's execution and analyse the correlation among the counts of various monitored events by continually learning the newly encountered program data with self-supervision while continuing to maintain reasonable overhead, unlike the supervised and unsupervised approaches which mostly suffer from labelled dataset and misclassifications on encountering new data [10], [11], [12], [14], [16]. We have considered multivariable time series data and found a correlation among them through graph attention networks, which have proven to be very useful in concluding the presence of an anomaly unlike a few recent studies on univariate time series analysis with supervised learning techniques [11], [13], [45].

The self-supervised learning approach of APPARENT continuously learns and adapts to new data. The model does not solely rely on labelled training data but also leverages the inherent structure and patterns within unlabeled data. This enables APPARENT to identify not only newly encountered programs but also probable threats in new programs by recognising deviations from the learned representations of normal behaviour. The combination of GTM and GAT in APPARENT's architecture contributes to its generalization capabilities. While APPARENT has been evaluated on the EEMBC autobench benchmarks, its underlying principles and techniques can be extended to other application scopes with similar characteristics. The autobench benchmarks simulate real time processing tasks commonly found in automotive systems, such as engine control, signal processing, and actuator control, some of which have been illustrated in this work. These tasks often involve tight timing constraints, limited resources, and the need for high reliability, similar to many other embedded applications. Some related application scopes where APPARENT's usage can be justified are real-time anomaly detection in CAN controllers of modern vehicles, monitoring the health and behaviour of real-time control systems in robots and autonomous vehicles, enhancing the security of connected vehicles by detecting anomalies in communication networks and in-vehicle entertainment systems [46].

3 APPARENT

A complete flowchart abstracting the entire process flow of APPARENT is as shown in Figure 2 and described below in the following steps:

- Program Execution and Data Collection: A program is executed on the embedded device, and an on-chip debugger collects trace data during the execution, including HPCs and PC values.
- The collected trace data is transmitted to an *edge server* for analysis.
- **Block A: CPI and Pattern Analysis:** Compute Average CPI: The process begins by computing the average Cycles Per Instruction (CPI) for the executing program.
- The CPI profile is then analysed to identify phases ph (function calls) and peaks pk (conditional branches).
- Analyse Similarity using GTM: The identified phases and peaks, along with their corresponding PC values, are fed into a Generative Topographic Mapping (GTM) model. This model maps the PC values onto a latent space, clustering similar patterns together. By comparing the observed patterns with the expected patterns learned during training, the GTM model can identify deviations that may indicate threats.
- Verification Module: The results from the GTM analysis are further verified to reduce false positives and improve the accuracy of anomaly detection.
- **Block B: Time Series Analysis using GAT:** Multi-Variable Time Series Analysis: In parallel to Block A, the HPC data, which includes various timer values, is treated as a multi-variable time series.

- Graph Attention Network (GAT): The GAT is employed to analyse the relationships and dependencies among different HPC events and their temporal patterns.
- Forecasting and Reconstruction Model: The GAT model is used to forecast future HPC values and reconstruct the observed time series. Deviations between the forecasted and actual values, as well as discrepancies in the reconstructed time series, are indicative of anomalous behaviour.
- **Block C: Maximum Voting Decision:** The anomaly detection decisions from both Block A (GTM) and Block B (GAT) are combined using a maximum voting algorithm.
- Anomaly Detected? The final decision on whether an anomaly has been detected is made based on the combined results from the previous blocks.

3.1 Threat Model

Embedded systems are more vulnerable to risks because of the ease with which they can be updated and connected to the network—in particular, to malicious code injection. APPARENT detects abnormal program behaviour caused by code injection attacks in embedded systems. It focuses on detecting deviations in program execution resulting from injected code, regardless of the specific attack vector (e.g., buffer overflows, format string attacks). The threat model assumes that injected code will cause observable deviations in program behaviour, which can be captured and analysed by monitoring low-level hardware events.

In a buffer overflow attack on an embedded platform, an attacker exploits a software vulnerability by sending input data that exceeds the allocated buffer size. This allows them to inject malicious code into the program's memory, potentially overwriting function pointers or return addresses. The injected code then gets executed, causing deviations in the program's expected behaviour. APPARENT monitors low-level hardware events and detects these deviations, indicating the execution as potentially harmful. By focusing on the low-level hardware events and their deviations, APPARENT can detect a wide range of code injection attacks, including but not limited to buffer overflows. The system's ability to continuously learn and adapt to new data further enhances its effectiveness in identifying and mitigating potential threats for embedded systems in general.

3.2 Detection strategy

Three structural levels are typically identified in a program from the perspective of software architecture: (a) function call stage, which displays the relationship between function calls; (b) a fundamental-block Control flow graph CFG representing the internal CFG for each program function, and (c) the stream of instructions for each CFG [47]. Based on a hardware perspective, the architecture and performance of a processor can have an impact on how instructions are executed. For example, a processor's performance may be degraded by condition branches or multi-cycle function calls. Alternatively, the PC register being an indicator of a program's position in a code sequence, can also convey the instruction order in a CFG. This also assists in analysing the PC values contained in every CFG after the initial detection of the CFG and function/routine call based on the variation in the performance of the processor. Ultimately, a comprehensive assessment may reveal whether or not the system is compromised. The average CPI is measured in the proposed work as a processor performance parameter.

Figure 3 displays a diagram of our proposed abnormal program behaviour identification model. First, the mean Cycles per Instruction CPI value is computed using block A in Figure 3, which continuously reads data out of the PC register and timer counter as well as clock cycle information. The data displays the locations of routine calls and conditional branches that occur on a running program and is acquired sequentially from the mean CPI values in the peak (*pk*) and phase (*ph*) point detector modules, respectively.

In block A, a GTM-based intelligent self-supervised similarity analyser is proposed to detect abnormal program behaviour by learning the locations that were obtained and the PC sequence that corresponded to them. The intrusion-detected result is asserted by the GTM-based classifier if the PC patterns and phase data are different from a known program. In the ultimate step, the GTM results are verified by contrasting them with the expected property table (which includes the number of peaks in each phase and the related network node).

Our proposed block B in Figure 3 tries to detect anomalies through time series analysis generated from multiple timers (HPC values) at various exception levels of our hardware platform while a program executes by finding a correlation among various HPCs generated time series. Following this, we perform a majority voting (block C) after incorporating anomaly detection decisions from both blocks A and B respectively to finally decide upon the occurrence of an anomaly.

3.3 Module for Computing Average CPI

In this subsection, we are going to describe the computation of the mean CPI module in block A of Figure 3. A popular metric for evaluating the processor's efficiency is CPI, which displays the intricacy of executed instructions in a given period. Thus one can compute a processor's average CPI given in equation 1.

$$CPI = C/I \tag{1}$$

where C is the no. of cycles needed to complete I instructions and I is the entire set of instructions being executed. Modern debug facilities can easily access the CPI since the no. of cycles is determined by combining the time spent and the processor's highest clock frequency. An ARMv8-A processor running a program generates an average CPI profile as shown in Figure 4, with I and the highest frequency initialised to 2^{12}.

The program in Figure 4 has six distinct routines, all of which are called sequentially. The CPI value increases dramatically when a new routine is called, which results in a corresponding decrease in processor performance. This is primarily because, to execute the recently called routine, the PC register jumps to a different memory location as shown in Figure 10 (can see anyone sample among a to f), typically consisting of multiple multi-clock cycle instructions. The

mean CPI value is thus considerably altered. The mean CPI profile is determined by the complete set of executed instructions (I), which ranges between $1\ to\ n$, with n denoting the program length.

In a similar manner, values of CPI also vary within every routine. With an increasing number of instructions, it is less likely to extract more details of CPI, which implies that some potentially aberrant behaviour of the program under observation might go undetected. Fewer I may enhance the sensitivity at the cost of increased computational expense.

For example, if I is 1, all of the program's instructions will be scrutinised and it won't include any continuous patterns that would allow us to identify the features of the program under observation. Hence, in this manuscript, I is set to 2^{12}, which has been shown to provide a better trade-off between the proposed system's computational complexity and accuracy [3], [15].

The choice of I involves a trade-off between sensitivity and computational overhead. A smaller I increases sensitivity to individual instruction variations but also increases the computational burden. A larger I provides a more smoothed average, potentially missing subtle anomalies but requiring less computation.

The subsequent subsections present an automated technique for acquiring the peaks (i.e. branch conditions) and phase (i.e. function calls) of an executing program under consideration.

3.4 Module for Detecting Phases(*ph*) and Peaks(*pk*)

Finding the *ph* and *pk* locations within the mean CPI outline is the primary objective of this module. There are two sub-blocks: the peak and phase positions are localised using the local and global critical point localisers.

3.4.1 Computing local critical points

The local primary variation points of the mean CPI are localised using the local critical point localiser. Using the suggested approach, the *pk* value is located within a 1×3 window rectangle range after the absolute variations between adjacent items within the mean CPI profile are first computed.

Let's say CPI_{mean} array contains the averaged CPI profile in equation 1. The next step is to compute the absolute differences between the adjacent elements of CPI_{mean} and is extracted as:

$$D_{ab}(i) = |CPI_{\text{mean}}(d+1) - CPI_{\text{mean}}(d)| \tag{2}$$

where $1 \leq d < N$, N denotes the complete set of elements present in the array CPI_{mean}, where now D_{ab} contains the absolute variations between the array's neighbouring elements CPI_{mean}. This D_{ab} is used to compute the peaks and valleys. After obtaining D_{ab}, a 1×3 rectangular window is scanned over all the elements of D_{ab} and the following computation is performed as shown in the pseudocode in Algorithm 1 to find out the local peaks. The primary advantage of the above methodology is that it can adaptively determine the *pk's* without any fixed thresholding making it a valuable candidate for a wide variety of scenarios. It is even capable of detecting peaks having minor variances.

Algorithm 1: Peak identification module

Input:
i. An input array D_{ab} containing absolute differences between adjacent elements of CPI_{mean} obtained from equation 2.
Output:
$D'_{ab}(j) = l_1, l_2, l_3, ..., l_i$ where l_1 denotes the set of locations for peak i

```
for i = 1; i ≤ N_samples(D_ab); i++ do
    if D_ab(i−1) < D_ab(i) and D_ab(i) > D_ab(i+1) then
        D'_ab(j) = D_ab(i); /* the amplitude is noted in array D'_ab(j)*/
m_i = mean(D'_ab(j)); /* the mean of all elements of D'_ab(j) is m_i*/
for i = 1; i ≤ N_samples(D'_ab); i++ do
    if D'_ab(j) > m_i then
        l_i = j; /* The jth element is marked as peak*/
Return D'_ab(j)
```

Figure 5 is the resultant detection after applying the Algorithm 1 on Figure 4. So Figure 4 represents the average CPI profile containing six different routines/functions called in a sequence, leading to different PC profiles, which is evident from the graph morphology. Figure 5 identifies the location of the peak positions, i.e. the conditional branches in a program as indicated by red triangles. These markings localise the significant local variance points derived from the average CPI profile.

3.4.2 Computing global critical points

The global notable variation points, which show the positions of each phase, are localised from the average CPI profile using the global critical point localiser and are illustrated in Algorithm 2.

In step 1 we localised the points which are greater than the threshold equivalent to $th = (max(D_{ab}) + min(D_{ab}))/2$ as described. They denote the boundary points of every adjacent phase. Following this, step 2 finds the absolute differences of every adjacent boundary point and stores the one greater than a threshold t_n whose values depend on the training program phase length. We have set it to 64 as a tradeoff between the efficiency and complexity of our proposed approach. The results from lines 13 and 14 of the algorithm 2 are used to find out the absolute difference of the adjacent phase elements as shown and store those points above 2 or 0. The objective is to ensure that the overlapping boundaries are not presented in the following phases.

The result of this global critical point computation is shown in Figure 6 which is obtained after applying it to the data from Figure 4. The resultant peaks and phases that are obtained from algorithm 1 and 2 respectively are being processed to convert them to their corresponding PC profile *as an outcome of algorithms 1 and 2 shown in the following equation 3.*

$$\begin{aligned} D'_{ab}b &= I \times D'_{ab}(j) + 1 \\ D'_{ab}e &= I \times D'_{ab}(j) + I \\ D_h b &= I \times D_h(ph') + 1 \\ D_h e &= I \times D_h(ph' + 1) \end{aligned} \tag{3}$$

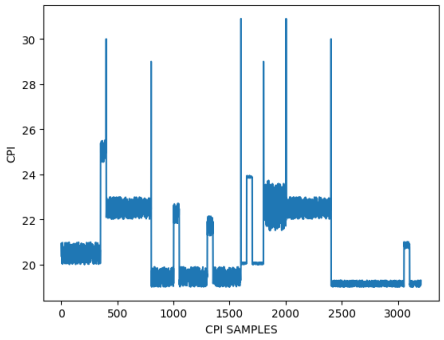


Fig. 4. Average CPI outline

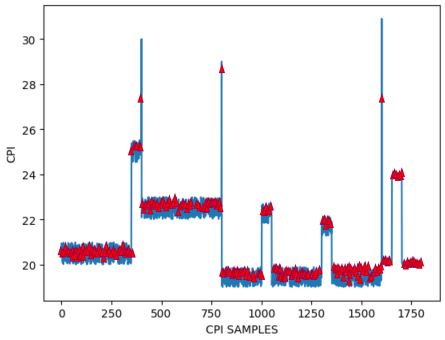


Fig. 5. CPI outline after processing through local critical point processing module

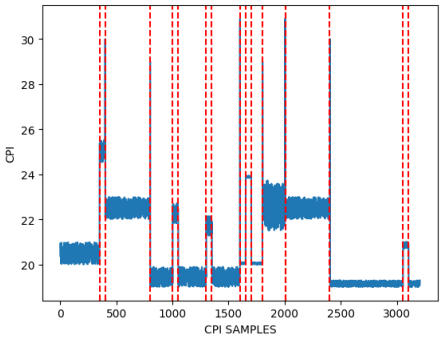


Fig. 6. CPI outline after processing through global critical point processing module

Algorithm 2: Phase identification module

Input:
i. An input array D_{ab} containing absolute differences between adjacent elements of CPI_{mean} obtained from equation 2

. **Output:**
$D_h(i) = l_1, l_2, l_3, ..., l_i$ where l_1 denotes the set of locations for phase i

Initalise $th = (max(D_{ab}) + min(D_{ab}))/2$.
/* Step1 */
for $i = 1; i \leq N_{samples}(D_{ab}); i++$ **do**
  if $D_{ab}(i) > th$ **then**
    $P'_{ab}(ph) = D_{ab}(i)$; /* $D'_{ab}(j)$ represents boundary points of every adjacent phase*/
    $ph++$;
/* Step2 */
for $ph = 1; ph \leq N_{samples}(P'_{ab}); ph++$ **do**
  $P''_{ab}(ph) = |P'_{ab}(ph) - P'_{ab}(ph+1)|$;
  /* t_n depends on training program phase length */
  if $P''_{ab}(ph) > t_n$ **then**
    $D_h(ph) = P'_{ab}(ph)$;
    $D_h(ph+1) = P'_{ab}(ph+1)$;
/* Step3 */
for $ph' = 1; ph' \leq N_{samples}(D_h); ph'++$ **do**
  $D'_h(ph') = |D_h(2ph') - D_h(2ph'+1)|$;
  if $D'_h(ph') > 2$ *or* $D'_h(ph') = 0$ **then**
    $D_h(2ph'+1) = D_h(2ph') + 1$;
Return $D_h(ph')$

where I denotes the complete set of instructions to compute the mean CPI. $D'_{ab}b$ and $D'_{ab}e$ are the beginning and end of the PC outline meant for j^{th} pk and D_hb and D_he denotes phase beginning and end positions for the ph^{th}. Figure 6 highlighting the red lines denotes the global critical points as the phase positions indicating the functions calls within a subroutine. Generally, function calls are not frequent in a program and hence are localised with the significant global variance as highlighted and are computed from the average CPI profile. These obtained locations are utilised to choose appropriate PC patterns to train and validate for the similarity analyser.

3.5 Module for Analysing Similarity using Generative Topographic Mapping (GTM)

After receiving the PC values, the peaks and the phases, we propose a Generative Topographical Mapping (GTM)–based similarity analyser (model size 73 KB approx) to recognise between known/trained and unknown/unlabelled running programs. The process of classification and recognition is divided into two main levels: the PC register pattern and the function/routine call level. At each level, the uniqueness of the program being executed is determined by measuring each phase and peak. Any appreciable variation indicates that there are fewer function calls, different function call characteristics, and a PC signature from the original program, that may indicate a probability of an abnormal behaviour. The primary benefit of the suggested similarity checker is that it controls the two levels of recognition and classification: the *pk* and *ph* and the PC sequence level. After statistical analysis of the *ph* and *pk* levels, the associated PC sequences are categorised in GTM. As a result, it is extremely unlikely for malicious codes to have a similar PC sequence as the original program, even if they share similar phase and peak information.

The pk and ph are effective markers since they act as indicators of program behaviour representing significant shifts in the CPI profile, corresponding to conditional branches and function calls. These are fundamental building blocks of program execution flow. Changes in their patterns can indicate deviations from the expected behaviour, even for unknown software. While the exact CPI values might vary somewhat due to hardware variations, the relative positions and patterns of peaks and phases remain relatively stable for a given program. This makes them robust indicators of program behaviour, even across different execution environments.

For normal programs, the pk and ph patterns should align with the expected behaviour learned during training. However, malicious software, even if designed to mimic legitimate behaviour, is likely to introduce anomalies in these patterns, due to the nature of its injected code. These anomalies can be detected by comparing the observed pk and ph patterns with the expected ones. It is to be noted that the pk and ph analysis is not meant to be a standalone anomaly detection technique. It is used in conjunction with other methodologies, such as PC sequence analysis and HPC time-series analysis, to provide a more comprehensive view of program behaviour and improve the overall accuracy of anomaly detection. The use of CPI profiles and the analysis of peaks and phases for program analysis and anomaly detection is not entirely new. Similar techniques have been explored in other works, primarily in the context of performance analysis and optimisation [48], [49]. We have

extended the concept of pk and ph analysis by incorporating it into a self-supervised machine learning framework that combines GTM and GAT. This allows for a more sophisticated and adaptable anomaly detection system that can handle unknown programs and distinguish between normal and malicious behaviour as described in the following subsections.

3.5.1 Data mapping to GTM

Our classifier has been designed on the basis of GTM [50]. We leverage the unique feature of GTM to distinguish the PC values and map them to a user-defined number of clusters. This is efficient enough to analyse and find the similarity between known and executing programs using PC patterns. It is to be noted that the total number of clusters should be equivalent to the total number of benchmark routines we are dealing with.

We have carefully chosen the number of clusters and set it to 16 as we have 7 different routine calls in the testing database. To be more precise, we take an embedded program's static properties and use them to enforce acceptable program behaviour during runtime.

The PC sequences are a collection of N-dimensional vectors, in which I denotes the set of instructions that have been executed per vector N. Fixing the correct value of N is very important and it should correspond to the minimum number of program counter values for accurate analysis. A large value of N may affect the application performance by adding more performance overhead and very little value may affect the distinguishing capability among the applications. The magnitude of N reflects appropriate behaviour for a program with carefully chosen properties indicating unaltered execution and is not likely to be breached when a program is hacked. The value of N used in this study has been determined to be 2^{12} after a series of empirical experiments and an analysis of the test data. We take the extracted values of the PC obtained from the peaks and phases as expressed in equation 3 and use it as input to the GTM model for initial training followed by verification.

Following the training procedure, we map every GTM node to the corresponding benchmark routine and subroutines. In our study, each GTM node has been labelled with a vector quantisation procedure [51] in the following manner: a) Every training data is assigned a label which acts as an identifier of the routine containing the training dataset, b) The winner GTM node in the latent space is found out having the least distance with the labelled input data, c) We maintain for every data input vector, the corresponding application label, the GTM node and the calculated distance. For applications which utilise similar address space, the distance is kept constant as a tiebreaker.

3.5.2 Finding statistical parameters

The initial training program involved counting a set of input vectors linked to each GTM node say n, followed by the computation of the group's mean value, standard deviation, and minimum and maximum distances expressed as:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} c^i \tag{4}$$

$$c_{min} = \mu - (1+\epsilon)\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(c^i-\mu)^2} \tag{5}$$

$$c_{max} = \mu + (1+\epsilon)\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(c^i-\mu)^2} \tag{6}$$

where c represents the list of distances and ϵ (empirically set to 3%) indicates the errors in the standard deviation for including quantisation errors during computation.

Following this, we propose to construct a statistical table Tb_{ph} for the ph phase containing the information of the attributes like (the minimum c_{min}, maximum distances c_{max}, the nodes associated with the input data vectors and the standard deviation). Every phase has an associated statistical table based on the same principle. During the inference, every input vector in the test set is assigned to a GTM node having the minimum distance. Assuming ξ^i represents the input data vector mapped to a GTM node ζ^i having distance c^i, our proposed methodology measures the maximum (C_{max}) and minimum (C_{min}) distances of the $\zeta^i th$ GTM node with the distance c for every lookup table, before deciding upon the phase to which the input vector belongs. A successfully assigned phase input vector should possess the two prerequisites namely a) $c_{min} < c^i < c_{max}$ where c_{max} and c_{min} are the maximum and minimum distances of the GTM node ζ^i at ph phase, b) As a dominant GTM node in the ph phase, the $\zeta^i th$ node in the initial lookup table indicates that its occupancy is higher than 5% of the total number of input vectors.

The phase numbers are reflected in the labels of the successful candidate GTM nodes. Otherwise, for an untrained/unknown input data vector, the candidate is marked 0. As a result, the known/trained program's ph ought to comprise an assortment of trained ph numbers, with the dominating phase no. serving as a signal for the phase's outcome. Another lookup table Tb'_{ph}, which has the same kind of data as Tb_{ph}, is created after the results of each phase are known. The next step involves similarity verification of these generated tables.

3.6 Verification module

The purpose of the verification module as shown in block A of Figure 3 is to verify the findings of the GTM analyser. With this analyser, the majority of the input vectors can typically be categorised. To enhance the overall performance of the categorisation, a global verification stage is required, because the program's trace might not always match the initial training program precisely due to varying circumstances. Generally speaking, there are two categories that the GTM analyser's results could fall into known/trained and untrained/unknown samples. The GTM analyser provides the possible phase number *ph* for known samples and marks the unknown/unidentified samples with '0', thereby handling the two cases independently. The flowchart is displayed in Figure 7. The primary task of the flowchart is to verify the computed similarity (in Algo. 3) between the testing/verifying and the original statistical lookup tables with their phases ph' and ph respectively. The flowchart initiates with its phase check, which if it happens to be

Algorithm 3: Similarity computation for the lookup tables

Input:
i. Tb_{ph}, $Tb'_{ph'}$ are the two look up tables of initial phase ph and validation ph' correspondingly.
Output:
i. $Csim$ similarity measure within the two phases ph and ph'

Initialise $Csim = 0$.
The look-up table Tb is sorted in the decreasing order of the GTM node occupancy.
/* Initiate searching in the look-up table Tb_{ph} */
forall *GTM nodes listed in table* Tb_{ph} **do**
  if *jth node coverage* $> 5\%$ **then**
    $A(p) = j$; /* the no. of GTM nodes is noted in array A*/
    $p = p + 1$;
/* Initiate searching in the look-up table $Tb'_{ph'}$ */
forall *GTM nodes listed in table* $Tb'_{ph'}$ **do**
  if *jth node coverage* $> 5\%$ **then**
    $A'(p) = j$; /* the no. of GTM nodes is noted in array A'*/
    $p = p + 1$;
$I \in A \bigcap A'$; /* the intersection between A and A' is I */
if $len(I)/len(A) > 80\%$ **then**
  phase $ph \sim ph'$
  $Csim$=1;/* non anomalous */
else
  phase ph ! $\sim ph'$
  $Csim$=0;/* anomalous */
Return $Csim$

nonzero, goes for further checking with the original and testing table phase data. If they are similar then the original phase number ph is assigned and flow terminates as shown in the left-hand side box. Else verifying phase is initialised with 1 and the flow moves to the second dotted box on the right-hand side. It then compares with the maximum phase of the original table. If it satisfies, then it goes for further checking with the original and testing table phase data, and if it matches, it is assigned the phase ph' and is incremented and iterates with the next level of checking with the maximum phase of the original table. Else the phase number is marked unknown.

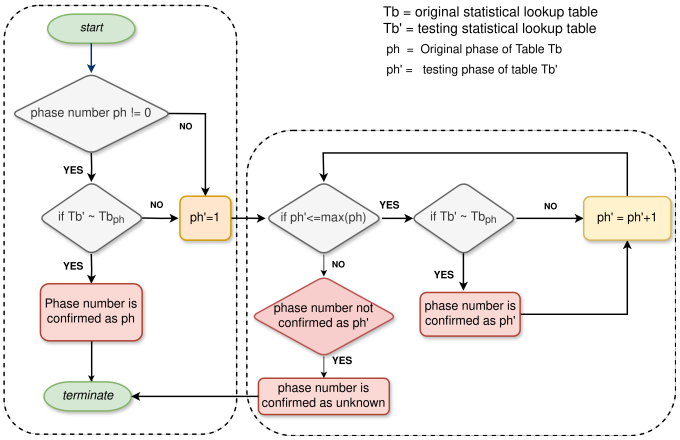


Fig. 7. Flowchart of the verification module (last module of block A in Fig. 3)

To verify the similarity measure between the verifying and original lookup tables namely Tb' and Tb respectively, the two tables' corresponding GTM nodal histograms are used. Algorithm 3 illustrates the pseudocode for computing the similarity between the lookup tables. The variation between the peaks pk in the verifying phase ph' w.r.t the original phase ph has been computed following a comparison of the lookup tables. Phase number ph is confirmed once the variation drops below 10% of the entire set of peaks present within the original phase. Generally speaking, the GTM analyser could determine the similarity between two input sets of vectors (as peaks) locally. It cannot, however, indicate a collection of peaks globally (as phases). This issue is resolved in the verification phase.

3.7 Using multiple variable time series analysis to detect anomalies

Besides block A we also designed block B as shown in Figure 3 to detect anomalies through multivariable time series analysis. Each benchmark program when executed generates various physical and virtual timer values at various exception levels on the embedded processor. Each timer variables (HPC) are taken as an individual feature variable and the correlations among the multiple feature variables (i.e multiple timers) and the time series dependencies are modelled simultaneously using Graph Attention Networks (GAT) so that any discrepancies identified while encountering any unknown/untrained routine, can be identified.

The problem is defined as follows: The input of multiple variable time series values generated while executing a program is $td \epsilon R^{T_m \times f}$, where T_m, f and R denote the highest timestamp length, the number of input features and real number respectively. Generally, to deal with longer time series, we take a fixed input length sliding window T_w. As an output vector of the multiple variable times series, $q \epsilon R^{T_w}$ is generated with $q_i \epsilon (0, 1)$ denotes the presence of an anomaly in the i^{th} timestamp. To tackle this issue, we first model the temporal dependencies and inter-feature correlations using two graph attention networks operating in parallel. After that, we use a Gated Recurrent Unit (GRU) network to identify long-term dependencies in the data sequence. We additionally utilise the potential of models based on reconstruction and forecasting by working together to optimise an integrated objective function.

Again referring to block B of Figure 3, we initiate the process by taking input from the various time series data being generated while a single benchmark program is being executed (shown in Figure 15)(a) and subject it to normalisation followed by single dimension convolution module (kernel size 5) to preprocess the first layer for extracting high-level features from every time series variable input [52]. We normalise it using the maximum and minimum values present in the training labelled dataset respectively on the input data say td given by:

$$\widetilde{td} = \frac{td - min(td_{label})}{max(td_{label}) - min(td_{label})} \tag{7}$$

This is followed by two parallel graph attention (GAT) [53] layers which process the outputs of the single-dimensional convolution layer, highlighting the connections between various features and timestamps. We proposed to

design a graph as illustrated in Figure 8 and represent our solution in a way where every node or vertices v_i is represented as the HPC parameter, i.e the various timers td_i in our example. We map each timer data td as a *single feature variable and map it to the vertices or nodes of the graph* where $v_i = (v_1, v_2, ...v_n)$ are the n nodes. The edges e_{ij} connecting the vertices determine the connections in terms of bonding among them. While computing the response for every node (say q of node v_7) in the graph, all of its neighbourhood node contribution is taken into account.

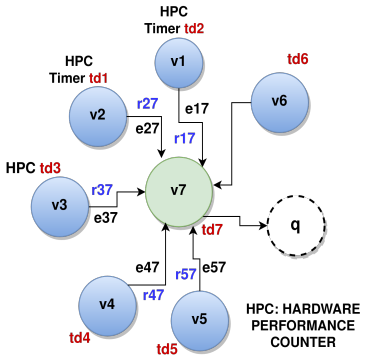


Fig. 8. Graph attention layer, $v_i \rightarrow td_i$ and $e_{ij} \rightarrow r_{ij}$ denotes the vertices (used timers as HPC in our study) and edges (relationship among timing events), dashed node q is the resultant

The resultant output for every node is expressed as:

$$q_i = \sigma(\sum_{j=1}^{N} r_{ij} td_j) \tag{8}$$

where σ denotes the sigmoid activation function, r_{ij} mapped to edges e_{ij} is the measure of the attention score which denotes the contribution of adjacent nodes(timers td_j in our study) from i to j and N is the total number of neighbourhood timers of j. Therefore, the attention score r_{ij} is given as:

$$r_{ij} = \frac{exp(Act_{ij})}{\sum_{l=1}^{N} exp(Act_{jl})} \tag{9}$$

$$Act_{ij} = LeakyReLU(TP^T.(td_i \bigoplus td_j)) \tag{10}$$

where $TP \epsilon R^{2d}$ a column vector consisting of trainable parameters and d denotes the feature vector dimension of every timer node. LeakyReLU represents a nonlinear activation unit. We exploit this property of the graph attention mechanism to compute the feature and time-oriented graph attention network.

3.7.1 Proposed feature and time-oriented GAT model

In the *feature-based GAT* module (model size 152 KB) we consider the multiple variable time series as an entire graph, and each graph node indicates a single feature (a particular timer data). Every edge connecting the nodes denotes the relationship between the adjacent feature nodes. Graph attention operations can thus be used to carefully capture the relationships between neighbouring nodes.

In the *time oriented GAT* module (model size 94 KB) we record time-series temporal dependencies. We view a sliding window's timestamps as a single, complete graph. In practical terms, the feature vector at timestamp t is represented by a node td_t, and all other timestamps in the current sliding window are represented by the nodes that surround it. Following this we merge the output representations obtained from the two GAT layers and the single-dimensional convolution layer (computed with equation 7 and 8), then feed it into a 32 unit (GRU model size 552 KB) layer expressed as:

$$q_i(feature) + q_i(time) + \widetilde{td} \rightarrow GRU$$
$$GRU \rightarrow tg \tag{11}$$

Time-series sequential patterns are captured using this GRU layer and generate the output tg. The resultant of the GRU layer is fed in parallel to both the *forecasting* and *reconstruction* model (Fig. 3) by computing an integrated optimisation target, as described in the next subsection.

3.7.2 Proposed forecasting and reconstruction model

The reconstruction model (model size 1.4 MB) is designed to learn the marginal data distribution by gaining knowledge of a latent representation of the complete time series, whereas the forecasting model concentrates on single-timestamp prediction. While training both the model parameters are simultaneously updated. During training the entire loss L_c is computed as the loss of individual models and expressed as:

$$L_c = L_{rec} + L_{forc} \tag{12}$$

where L_{rec} and L_{forc} represents the reconstruction and forecasting module losses respectively.

The forecasting module is designed using 2 fully connected layers (model size 3 MB approx) and is responsible for computing the next predicted time stamp and its corresponding loss is computed as Root Mean Square Error (RMSE):

$$L_{fore} = \sqrt{\sum_{i=1}^{f} (tg_{n,i} - \hat{tg}_{n,i})^2} \tag{13}$$

where the subsequent timestamp is represented by tg_n for the present input $tg = (tg_0, tg_1, ..., tg_{n-1})$, where $tg_{n,i}$ denotes the i^{th} feature at tg_n and $\hat{tg}_{n,i}$ the predicted value of the forecasting module, f set of input features.

Learning a minimal distribution of information over a latent representation is the goal of the reconstruction model. To describe an observation in the latent space in a probabilistic way, we use a tiny Variational Auto-Encoder (VAE) [54]. The model captures the data distribution of the complete time series by treating the time-series values as variables. With input data tg, it is intended to be reconstructed using a conditional probability distribution $p(tg|l)$ with l denoting the latent space vector representation and a recognition expression $r(l|tg)$ the reconstruction loss L_{rec} is computed as:

$$L_{rec} = -E_{r(l|tg)}[logp(tg|l)] + KL_d(r(l|tg)||p(l)) \tag{14}$$

where the first expression computes the anticipated negative log-likelihood of the input and the second one computes the KL divergence between the two distributions. Finally, the model inference is computed after considering the integrated optimisation target computing the predicted values of forecasting and reconstruction models.

For every feature, we compute an inference value I_i, and the final inference value P_{value} is the sum of all the features expressed in equation 15. If a timestamp's matching inference score is higher than a predetermined threshold, we classify it as an anomaly. To determine the threshold automatically, we employ [55] which is expressed as:

$$P_{value}(L_{rec}, L_{fore}, \lambda) = \sum_{i=1}^{f} I_i = \sum_{i=1}^{f} \frac{(tg_i - \hat{tg_i})^2 + \lambda \times (1 - p_i)}{1 + \lambda} \tag{15}$$

where $(\hat{tg_i} - tg_i)^2$ represents the forecasting error as the absolute deviation between the predicted and present actual value, and $(1 - p_i)$ denotes the likelihood of coming across an anomalous value by the reconstruction model and λ a hyper-parameter which combines the reconstruction probability and the forecasting error, is chosen by a grid search on the testing dataset.

For the reconstruction loss, the performance indicators are Negative log likelihood (NLL) and KL divergence. NLL measures how well the reconstructed data matches the original input. A lower NLL indicates a better match. KL Divergence measures the difference between the learned distribution of the latent representation and a prior distribution (often a standard Gaussian). A lower KL divergence means the learned representation is closer to the expected distribution, promoting generalization. The overall L_{rec} is a weighted combination of NLL and KL divergence that can be tuned to prioritize either accurate reconstruction (NLL) or generalization (KL divergence).

The performance indicators of forecasting loss are RMSE and Mean Absolute Error (MAE). RMSE measures the average difference between predicted and actual values. A lower RMSE indicates better prediction accuracy. MAE on the other hand measures the average prediction error but is less sensitive to outliers than RMSE. The choice between RMSE and MAE depends on the specific application and the impact of outliers. Having used RMSE we have analyzed prediction errors over different time horizons that revealed how well the model captures long-term dependencies. The integrated objective function typically combines both losses as a weighted sum. Based on application an optimal balance is necessary. If accurately capturing the global data structure is crucial, a higher weight might be given to L_{rec} else if predicting future trends is more important, a higher weight might be given to L_{fore}.

3.8 Proposed maximum voting decision algorithm

Once we compute the decision of both blocks A and B respectively, we compute the majority voting to finally conclude the occurrence of an anomaly using another block C as shown in Figure 3. and the pseudocode is described in Algorithm 4. It illustrates the maximum voting decision algorithm. It receives the input of the resultant decision of two blocks A and B (described in Figure 3), namely the $Csim$ and P_{value} respectively regarding the occurrence of an anomaly. The occurrence of an anomaly is assumed to be a negative class and a healthy state is considered a positive class as an output of the algorithm. The algorithm iterates through all the routine calls for a program under execution initiating in step 2. The resultant blocks A and B are assigned to variables $c1$ and $c2$ respectively (steps 3 and 4). If both $c1$ and $c2$ are proven to be anomalous, then the particular routine is designated as unknown (steps 5 and 6) and the true negative variable is incremented (step 9), else the false negative count is incremented (step 12). In the end, these counts are returned for further accuracy, precision and recall computation in Table 3 (step 13). Following this in the next subsection, it is necessary to discuss the training and testing dataset generation from the EEMBC dataset.

Algorithm 4: Maximum Voting Decision Algorithm

Input:
i. Decision from block A ($Csim$).
ii. Decision from block B (P_{value}).
Output:
i. Return tn_{count}, fn_{count}

1 Initialise $c1$,$c2$; The sample program Pg consists of N routine calls.
2 **for** $i = 1; i < N; i++$ **do**
3 $c1 = Csim$;
4 $c2 = P_{value}$;
5 **if** $(c1 \& c2) == anomalous$ **then**
6 $N(i)$ = unknown_program;
7 **if** $N(i) == unknown\_program$ **then**
8 /* increment true negative */
9 $tn_{count}++$;
10 **else**
11 /* increment false negative */
12 $fn_{count++}$;
13 **Return** tn_{count}, fn_{count} for computing Acc,$Prec$, Rec in Table 3.

3.9 Training and testing dataset creation using random generation of benchmarks

The five benchmark routines from the well-known EEMBC benchmark suite's automotive package shown in Table 1 are combined to create a new program such that *each benchmark* can be identified as a distinct *routine call*, within the newly created program to train using all five of them. To produce sufficient training samples, the new program is executed four times. A random number module is utilised to call the benchmarks (as routines) randomly from the validation samples during testing to confirm the proposed system's performance in scenarios with dynamic fluctuations (i.e., interruption inputs, different program flow, etc.). The proposed system is tested for complex test samples in a range of scenarios by randomly selecting the benchmarks and creating a new program by randomly combining the benchmarks using a random function call generator. As a result, each time the new program runs, a different function call sequence is used. Furthermore, through the random mixing of the testing program with various function calls during the embedded system's run-time, the testing methodology can also be used to confirm the proposed system's performance in scenarios with dynamic variance. The testing case pattern is generated using three categories described in subsections 3.9.1 and 4.1.

For example, a program consisting of f_n distinct routine calls, random no. r is produced initially such that $1 < r < f_n$. This generated random no. activates the corresponding routine call (for example, *bitmnp* will be called if $r = 2$).

3.9.1 Utilising the benchmark routines as test suite

This study used seven benchmark algorithms from the well-known EEMBC benchmark suite's automotive package [56]. Referring to Table 1, the five benchmarks are used for training the AI models namely GTM and GAT and the remaining two are used for testing purposes. The seven benchmarks each have different parameters and serve different purposes. Taking the example of pulse width modulation (PWM), (*puwmod*), the fifth benchmark in Table 1. This EEMBC benchmark mimics a situation where a PWM signal proportionate to an input drives an actuator. The *phase signals* regulate the motor direction and the PWM signals offer proportional velocity control. Overall, they are appropriate to test subjects for the suggested experiments because they have comparable sub-routines in addition to differing complexities and characteristics.

3.9.2 Applicability to embedded devices

APPARENT is designed to be adaptable to various embedded devices. It analyses low-level hardware changes, like program counter values and cycles per instruction, which are common across embedded architectures. While specific model parameters may be fine-tuned, the underlying principles and methodology are generalisable. The self-supervised learning approach allows continuous adaptation to new data, including data from different devices.

To handle hardware heterogeneity, APPARENT focuses on relative patterns and relationships among HPCs rather than absolute values. The model analyses changes in CPI over time and patterns of peaks and phases, which are more consistent across devices. The self-supervised learning approach also enables adaptation to specific device characteristics. APPARENT can be deployed in several ways and would allow the effective detection of anomalies across diverse embedded systems.

- Centralised Model: A single model can be trained on data collected from multiple devices of the same type or even across different types. This centralised model can then be used to analyse data from new devices, leveraging its broad understanding of normal behaviour to detect anomalies.
- Device-Specific Fine-Tuning: The centralised model can be further fine-tuned for specific device types or use cases by incorporating device-specific data during training. This can improve the model's performance and accuracy for those particular devices.
- Ensemble of Models: An ensemble of models, each trained on data from a specific device type, can be used to analyse data from new devices.

4 Experimental Setup

We have considered an embedded system built upon an ARMV8.2 Morello prototype board equipped with ARM MORELLO SoC, with a prototype architecture built with a new, experimental, out-of-order CPU based on the Neoverse-N1 processors. Morello architecture is becoming a popular choice for autonomous vehicles and other IoT devices [57]. Our target platform consists of multiple peripheral interfaces like Ethernet ports, USB interfaces, and debug trace ports and comes with a DSTREAM-PT debug probe capable of high-performance debugging and tracing, suitable for fast downloads and can also adapt to JTAG clock rates. Up to 300 MHz DDR (600 Mbit/s per pin) has been used for 32-bit wide trace capture [58].

TABLE 1
Autobench Benchmarks used for training and validation [56]

sl no	Benchmarks	Description	Parameters
1.	Angle to Time conversion (a2time)	Measuring engine speed	NUM_TESTS: 1000 TENTH_DEGREES:3600
2.	Bit Manupulation (bitmnp)	Decision making based on bit values	NUM_TESTS: 256 INPUT CHARACTERS: 50 CHARACHTER_COLUMNS:10
3.	Inverse Discrete Cosine Transform (idctrn)	Transform on input data matrix set using 64-bit integer arithmetic	NUM_TESTS: 4096 COS_SCALE_FACTOR: 512 COS_SCALE_EXP: 24
4.	Road Speed Calculation (rspeed)	Road speed based on differences between timer counter values	NUM_TESTS: 1000 SPEED_SCALE_FACTOR: 36000
5.	Pulse Width Modulation (puwmod)	The PWM signal drives an actuator	NUM_TESTS: 4840 maximum_phase: 40
6.	Table Lookup and Interpolation (tblook)	selective data points are stored interpolation is performed between them	NUM_TESTS: 256 NO_of_X_ENTRIES : 100 NO_of_Y_ENTRIES : 100
7.	Tooth to spark (ttsprk)	performs real-time processing of air/fuel mixture and ignition timing.	NUM_TESTS = 500 cylinders: 16

Referring to Figure 9, the experimental setup consists of a host PC which is connected to the Morello prototype board under test, employing a DSTREAM-PT trace device, which probes the runtime data from the executing program by the ARM processor residing in the Morello hardware prototype platform. This runtime data from various functions and variables of the programs is traced by the trace device through the trace port and is sent to the Host PC.

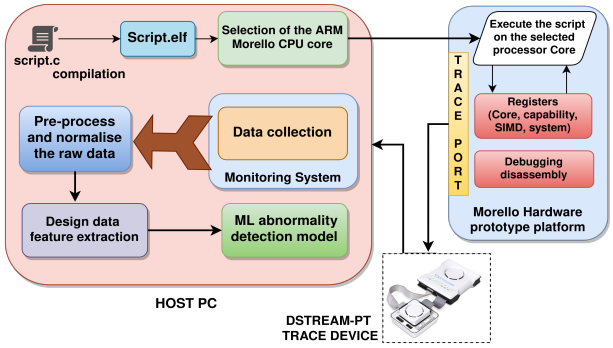


Fig. 9. APPARENT Experimental setup

The trace data once obtained by the host PC is subjected to preprocessing and normalization for further feature extraction and classification between known and unknown (anomalous) programs using AI models in real time. Since in practicality, most IoT platforms run without OS, we execute bare-metal-standalone benchmark routines on the Morello prototype board, capture the trace information and perform further analysis in the host PC, thereby making our system a viable solution to be deployed in edge servers tier which lies very close to the IoT device and can save the costly cloud server communication (Fig. 1).

4.1 Validation inputs

Using the random generation of benchmarks as routine calls we generate around 300 test programs and have divided them into 3 groups as itemised below:

- *Category 1*: programs using the original routine-calling order: Under this group, programs follow the identical order as used during the training phase and are firm in the call sequence (vi. Fig. 10 (a) and (b)).
- *Category 2*: programs using randomly generated routine-calling order: Under this group, programs follow the random pattern of routine calls, however, all the randomly called benchmarks are called from the 5 *known* training examples (Fig. 10 (c, d)).
- *Category 3*: programs using randomly generated routine-calling order: Under this group, programs follow the random pattern of routine calls, however, all the randomly called benchmarks as routines are called from the *known* training examples and the 2 *unknown* benchmarks (Fig. 10 (e) and (f)).

The program numbers used from these 3 groups are 90, 90 and 120 respectively. The first group in the experiment is meant to mimic situations in which the embedded system is left unmodified, like programs that run straight out of the factory. The second category, which includes programs having valid credentials for execution on the experimental embedded platform, is used to mimic the conditions of an embedded system operating normally. The final category is utilised for modelling compromised platforms that have untrained/unknown programs installed; for instance, a use-after-free attack might trigger the system to run some unidentified programs. Therefore, we have tested the proposed threat model with all 3 groups of datasets. Figure 10 illustrates the PC profiles of the trained examples (labelled as 1,2,3,4,5) and unknown/testing benchmark (labelled as $u1, u2$ within a square box) examples respectively.

The PC addresses and every benchmark profile vary marginally, despite having identical benchmark codes and sequences, as shown in Figs. 10(a) and (b). Particularly, when the benchmark sequence is mixed at random (as in Fig. 10(c), (d)), the generated profiles for the PC are completely different from each other. This may be useful when examining the false negative rate of the trained GTM+GAT analyser. Figures 10(e), (f) show how the profiles of the unknown routines marked as $u1$ and $u2$ resemble those of the known routines ”1” and ”5” correspondingly. These unknown routines (potential injected code) have PC profiles that resemble known routines. These profiles are meant to imitate possible attacks that try to replicate the phase information and peaks of the actual programs. Various programs with comparable profiles can be further explored utilising the false/true positive as well as negative rates from the already learned GTM+GAT analyser.

4.2 APPARENT Building blocks

HPCs and CPI modules: Our chosen HPC module set consists of various physical and virtual timer count values at different exception levels while a particular benchmark is executing. This indicates various information like the execution time of multiple CFGs within a benchmark routine, the starting and ending times and their interrelationships. Any deviations of these timings from the known trajectory of the multiple HPCs will denote an anomalous situation.

The CPI module computes the average CPI for each run after first extracting pertinent data from the program's

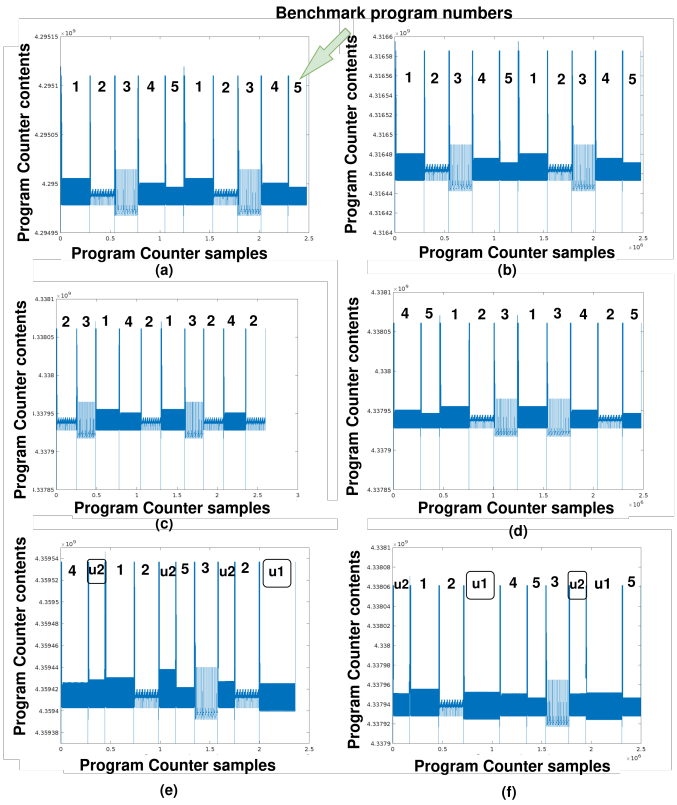


Fig. 10. (a and b) Samples of PC profiles from categories 1 and 2, (c and d) belong to category 2 and (e and f) belong to category 3. N.B $u1$ and $u2$ are the unknown benchmarks namely *tblook* and *ttsprk* respectively that haven't been used for the training and are used as routine calls.

tracing file. Every executed instruction's time tag and PC address are recorded in the program's tracing file. The only file containing the PC addresses is the one used by the GTM-based similarity analyser module. On the other hand, the CPI profile for the programs that are being executed is computed using the matching time tags. The number of instructions in this work is 4096. The average CPI for each of the 4096 instructions is computed using equation 1. The phases and peaks from the mean CPI are then localised by the *ph* and *pk* point detector. Finally, using 3, the acquired *ph* and *pk* locations are translated into the appropriate locations in the PC outline. GTM+GAT based classifier: The GTM-based similarity analyser block A (Fig. 3) receives an input vector with 1×4096 elements that are formed by choosing a serial of PC addresses based on the beginning and end locations of every peak. The GTM's maximum node count and iteration count are 50 and 2000, respectively. Following the training process, an estimated output for each peak and a statistical table for each phase are produced. During testing, the same procedure is carried out again. The verification module then makes use of the produced results.

The corresponding HPC data of each benchmark is fed into the GAT module in block B (Fig. 3). Some of the HPC data are like Program counter (PC) and PC with capabilities (PCC), cycles per instruction, and physical and virtual timer values. The corresponding events considered are primarily the instructions, branches, branch-misses, cache-references, cpu-cycles, bus-cycles, cpu-clock, cycles-ct, cpu-migrations, dTLB-loads, dTLB-store, branch-instructions, mem-loads, and mem-stores. The sizes of hidden dimensions of GRU,

TABLE 2
Effect of variation in λ on autobench benchmark with Accuracy(Acc), Precision($Prec$) and Recall(Rec)

λ	Acc	$Prec$	Rec
1	0.9120	0.786	0.9257
0.8	**0.9478**	**0.7934**	**0.9502**
0.6	0.9263	0.7842	0.9359
0.4	0.9221	0.7825	0.9311

variational autoencoder and fully connected layers are set to 32, 100 and 100 respectively. Our model is trained for a total of 100 epochs at an initial learning rate of 0.001 using the Adam optimiser. Verification and evaluation: This module implements the proposed APPARENT methodology described in Section 3. For the GTM algorithm in block A (Fig. 3), the final classification of each input program's peaks and phases is based on the verification results. There are two levels to the outcome of the final assessment: phase and peak levels. The top-level, ultimate report contains the database's contents in addition to the outcomes for each program. Similarly, the GAT algorithm in block B (Fig. 3), finds the correlation among various HPC events and models the temporal dependencies and inter-feature correlations using GAT. Any deviation from the program's expected execution trajectory signifies the probability of occurrence of an anomalous behaviour. Once we compute the decision of both blocks A and B respectively, we compute the majority voting to finally conclude the occurrence of an anomaly using another block C as shown in Figure 3, and the pseudocode is outlined in Algorithm 4.

TABLE 3
Performance of the autobench benchmarks

Benchmarks	Case 1 with only block A (Values in %)			(Case 2) with block B using attention network (Values in %)		
	Acc	$Prec$	Rec	Acc	$Prec$	Rec
a2time	42	93	42	96.6	98	98.2
bitmnp	37	91	39.2	97	99	98.1
idctrn	53.1	99.6	58	96	98.3	97.6
puwmod	92.5	95	95	97.3	99	97.5
rspeed	87	98.5	91.6	98	99.03	98.4
tblook	96.2	0	0	98.4	0	0
ttsprk	97.2	0	0	98.5	0	0
General Performance	72.5	97	67	98.46	99	98

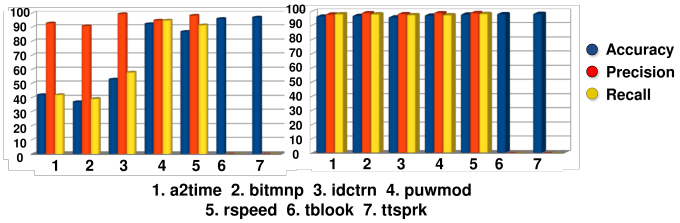


Fig. 11. Graphical representation of the performance of the autobench benchmarks, with x axis representing the seven benchmarks.

4.3 Results and observations

The experimental results obtained from GTM block A (Fig. 3) divide the program peaks and phases into distinct groups; known peaks (*pk*) and phases (*ph*) are given names that correspond to them, while unknown *pk's* and *ph's* are marked with the symbol "0" As a whole, block A has successful identification rates of 97.9% and 95.7% for 1140 phases and 175536 peaks, correspondingly. Furthermore, our proposed system achieves over 96.6% accuracy in identifying peaks of unknown/untrained programs not a part of the training dataset. The evaluated results are grouped by type of program as highlighted in the following subsections.

- *Category 1*: programs using the original routine-calling order with 90 (*known* and *same training order*) programs having 48238 peak values and achieving 95.9% accuracy.
- *Category 2*: programs using randomly generated routine-calling orders with 90 (*known*) programs having 66942 peak values and achieving 95.2% accuracy.
- *Category 3*: programs using randomly generated routine-calling orders with 120 (both *known* and *unknown*) programs having 60356 peaks and achieving 95.6% accuracy.

Similarly, the final results obtained from only block B are computed with the score value as illustrated in equation 15. The results have been evaluated with different values of λ and are shown in Table 2, with the highest score obtained with $\lambda = 0.8$. Finally, after we combine the results of both blocks A and B and perform a majority voting in block C (Fig. 3), we obtain the final results as displayed in the three use cases shown in Figure 12, 13 and 14 respectively. When the three categories of programs are employed for testing as described above, the overall accuracy increases after combining block B as observed in Table 3, due to a decrease in false negative rates.

The first category generally has higher accuracy (*Acc*), precision (*Prec*), and recall (*Rec*) rates due to the variety in test category complexity w.r.t the other two categories. The second and third categories, on the other hand, have comparatively lower *Acc*, *Prec* and *Rec* rates, unlike the first. Additionally, each verified program in categories 2 and 3 has a different type and duration, which results in a relatively higher variance in each program's resultant rates compared to the first. Table 1 shows that the database used is primarily made up of seven unique benchmarks, among which five fall into the training set, while the test set contains the remaining two. The *Acc*, *Prec* and *Rec* rates obtained for each benchmark are summarised in Table 3, with the graphical representation in Figure 11.

When block B is not used (Case 1 in Table 3), the *Acc* and *Rec* rates for the initial benchmark, named *a2time*, are significantly smaller. The rationale is that *a2time* and *tblook* are grouped in the same cluster because their distances from the sample GTM grid node are extremely similar. Since the test samples differ slightly from the training dataset samples, the *Acc* and *Rec* rates for the established benchmarks are also lower than the outcomes obtained after combining block B. Given that none of the groups contain any positive examples, the *Prec* and *Rec* ranks for the unknown benchmarks are zero both with and without block B computation.

As illustrated in Figure 15(a) the highlighted section after the 20^{th} sample displays that there is a sudden drift in the values of the multiple HPC timers which indicates

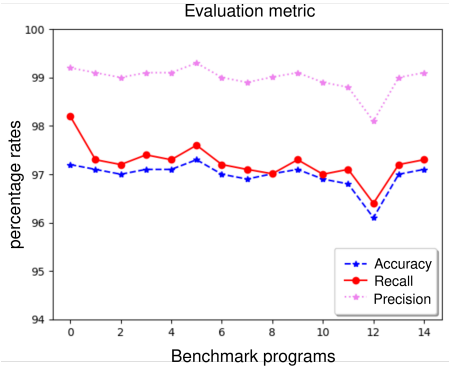


Fig. 12. Percentages of Accuracy (Acc.), precision (Prec.) and Recall (Rec.) for category 1

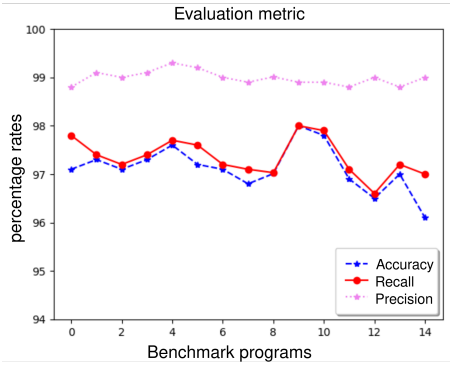


Fig. 13. Percentages of Accuracy (Acc.), precision (Prec.) and Recall (Rec.) for category 2

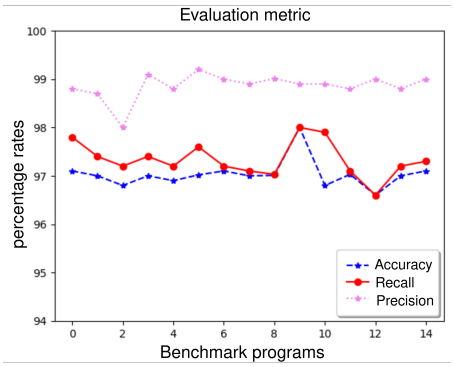
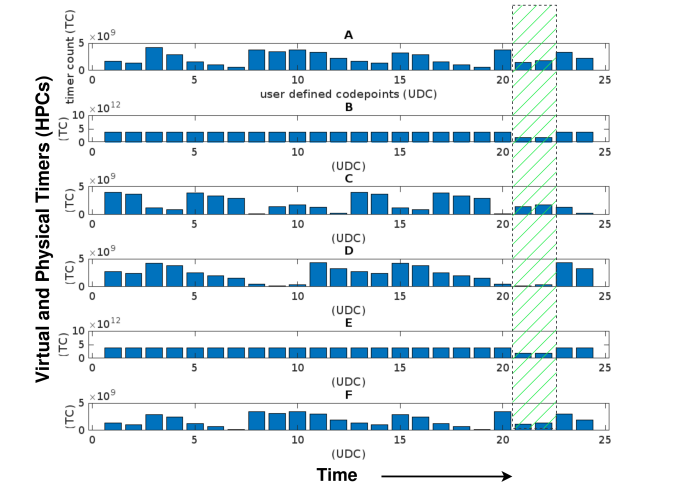
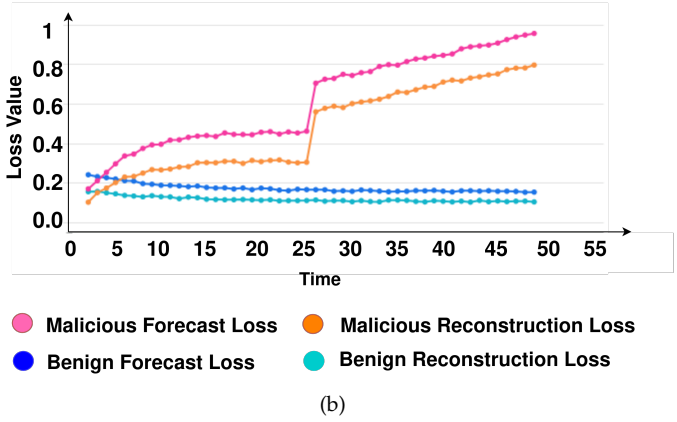


Fig. 14. Percentages of Accuracy (Acc.), precision (Prec.) and Recall (Rec.) for category 3



(a)



(b)

Fig. 15. (a) HPC timer value drift as highlighted when it encounters unknown/untrained program (x-axis timer values, Y-axis the user-defined codepoints time series (b) change in the loss curve with time for forecasting and reconstruction module respectively distinguishing between unknown benign and malicious programs.

that an unknown program, has arrived and this pattern is not known to the GAT module and its subsequent forecasting and reconstruction models. Our proposed methodology monitors physical and virtual HPC timers to identify these anomalies. This sudden drift in HPC timer values as highlighted, indicates data tampering and the presence of unknown code execution, resulting in a sudden change in the correlated timer values influencing the program's control flow. The self-supervised learning approach employed by APPARENT allows the model to continuously learn and adapt to new data. The model does not solely rely on labelled training data but also leverages the inherent structure and patterns within unlabeled data, thereby enabling APPARENT to identify anomalies in new programs by recognizing deviations from the learned representations of normal behaviour. Its reliance on low-level hardware variations, such as program counter values, cycles per instruction, and timer values, provides a fundamental understanding of program behaviour. By analysing these hardware events, it can detect deviations from the expected patterns, even for programs it has not encountered during training i.e. unknown programs. The plots in Figure 15(b) illustrate the progression of the loss over time for each learning model. For the unknown benign program, Reconstruction Loss maintains a relatively constant loss value throughout the period and Forecast Loss shows a decreasing trend in loss value over time. However, for an unknown malicious program, both the reconstruction loss and forecast loss exhibit a sharp increase in loss value, since the GAT couldn't find a stable learned correlation among the various HPC events.

5 Conclusion

In this manuscript, we propose a novel system design APPARENT, which identifies program characteristics by monitoring and counting the maximum possible low-level hardware events from HPCs that occur during the program's execution and analyse the correlation among the counts of various monitored events. To further utilise these captured events as features we propose a self-supervised machine learning algorithm that combines a GAT and a GTM to detect unusual program behaviour as anomalies to enhance the system security. Our proposed methodology takes advantage of attributes like program counter, cycles

per instruction, and physical and virtual timers at various exception levels of the embedded processor to identify abnormal activity. We have shown the efficacy of the proposed approach in various experimental scenarios. While experimenting with analysing the trace of debug data on the Host PC we faced transmission overheads in terms of large data volumes between the target board and the analyser running on the host PC, transmission delays due to heavy bus utilisation and also had to be selective in terms of leveraging appropriate communication protocol. We are investigating the use of lossless compression techniques to reduce the data volume transmitted and explore the use of different communication protocols that are optimised for low-latency and high-bandwidth data transfer. Finally, to reduce the data communication challenges further, we plan to develop this technology on the device itself.

ACKNOWLEDGMENT

This work is supported by the UK Engineering and Physical Sciences Research Council through grants, EP/X015955/1, EP/X019160/1 and EP/V000462/1, EP/V034111/1, EP/Z533749/1. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising.

REFERENCES

[1] J. Zhou et al., "Swarm intelligence based task scheduling for enhancing security for iot devices," *IEEE TCAD of Int.CAS.*, 2022.
[2] V. A. Memos, K. E. Psannis, and Z. Lv, "A secure network model against bot attacks in edge-enabled industrial internet of things," *IEEE Trans. on Indust. Inform.*, vol. 18, no. 11, pp. 7998–8006, 2022.
[3] X. Zhai et al., "A method for detecting abnormal program behavior on embedded devices," *IEEE Trans. on Inform. Forens. and Secur.*, vol. 10, no. 8, pp. 1692–1704, 2015.
[4] Q. Zhang et al., "Efficient anonymous authentication based on physically unclonable function in industrial internet of things," *IEEE Trans. on Inform. Foren. and Sec.*, vol. 18, pp. 233–247, 2022.
[5] Z. Pan, J. Sheldon, and P. Mishra, "Hardware-assisted malware detection and localization using explainable machine learning," *IEEE Trans. on Comput.*, vol. 71, no. 12, pp. 3308–3321, 2022.
[6] S. D. Paul et al., "Rihann: Remote iot hardware authentication with intrinsic identifiers," *IEEE IoT Journal*, vol. 9, no. 24, pp. 24 615–24 627, 2022.
[7] Z. Pan et al., "Hardware-assisted malware detection using machine learning," in *2021 Des., Autom. & Test Europe.(DATE)*. IEEE, 2021, pp. 1775–1780.
[8] Y. Singh, A. P. Kuruvila, and K. Basu, "Hardware-assisted detection of malware in automotive-based systems," in *2021 DATE*. IEEE, 2021, pp. 1763–1768.
[9] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—a state of the art survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–48, 2019.
[10] M. Bourdon et al., "Hardware-performance-counters-based anomaly detection in massively deployed smart industrial devices," in *2020 IEEE 19th Intern. Symp. on Netw. Comput. and Appli. (NCA)*. IEEE, 2020, pp. 1–8.
[11] L. L. Woo et al., "Early detection of system-level anomalous behaviour using hardware performance counters," in *2018 Des., Autom. & Test Europe.(DATE)*. IEEE, 2018, pp. 485–490.
[12] M. F. B. Abbas et al., "Hardware performance counters based runtime anomaly detection using svm," in *2017 TRON Symp. (TRONSHOW)*. IEEE, 2017, pp. 1–9.
[13] A. P. Kuruvila et al., "Time series-based malware detection using hardware performance counters," in *2021 IEEE Intern. Symp. on Hard. Oriented Secur. and Trust (HOST)*. IEEE, 2021, pp. 102–112.
[14] Y. Hu et al., "Care: Enabling hardware performance counter based malware detection resilient to system resource competition," in *2022 IEEE (HPCC/DependSys)*. IEEE, 2022, pp. 586–594.
[15] X. Zhai, K. Appiah, S. Ehsan, G. Howells, H. Hu, D. Gu, and K. McDonald-Maier, "Exploring icmetrics to detect abnormal program behaviour on embedded devices," *Journal of Systems Architecture*, vol. 61, no. 10, pp. 567–575, 2015.
[16] S. P. Kadiyala et al., "Hardware performance counter-based fine-grained malware detection," *ACM Trans. on Emb. Computing Sys. (TECS)*, vol. 19, no. 5, pp. 1–17, 2020.
[17] H. Wang et al., "Scarf: Detecting side-channel attacks at real-time using low-level hardware features," in *2020 IEEE 26th Int. Sym. on On-Line Test. and Robust Sys. Des. (IOLTS)*. IEEE, 2020, pp. 1–6.
[18] A. P. Kuruvila et al., "Hardware-assisted detection of firmware attacks in inverter-based cyberphysical microgrids," *Int. Journal of Electrical Power & Energy Systems*, vol. 132, p. 107150, 2021.
[19] C. Li et al., "Detecting spectre attacks using hardware performance counters," *IEEE Tran. on Comp.*, vol. 71, no. 6, pp. 1320–1331, 2021.
[20] A. Biswas et al., "Multi-granularity control flow anomaly detection with hardware counters," in *2021 IEEE 7th World Forum on Int. of Things (WF-IoT)*. IEEE, 2021, pp. 449–454.
[21] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings 17*. Springer, 2014, pp. 109–129.
[22] G. Michau and O. Fink, "Unsupervised transfer learning for anomaly detection: Application to complementary operating condition transfer," *Knowledge-Based Systems*, vol. 216, p. 106816, 2021.
[23] H. Sayadi et al., "Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis and classification," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.
[24] J. A. dos Santos et al., "Hierarchical density-based clustering using mapreduce," *IEEE Tr. on Big Dat.*, vol. 7, no. 1, pp. 102–114, 2019.
[25] J. Zhan et al., "Stgat-mad: Spatial-temporal graph attention network for multivariate time series anomaly detection," in *ICASSP 2022-2022 IEEE (ICASSP)*. IEEE, 2022, pp. 3568–3572.
[26] A. et al., "Dynamic task offloading and scheduling for low-latency iot services in multi-access edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 37, 2019.
[27] W. Jin et al., "Ulpt: A user-centric location privacy trading framework for mobile crowd sensing," *IEEE Trans. on Mob. Comput.*, vol. 21, no. 10, pp. 3789–3806, 2021.
[28] A. et al., "Robust malware detection for internet of (battlefield) things devices using deep eigenspace learning," *IEEE Tran. on sust. computing*, vol. 4, no. 1, pp. 88–95, 2018.
[29] K. D. Maier, "On-chip debug support for embedded systems-on-chip," in *Proc. of the 2003 Intern. Symp. on Circ. and Syst., 2003. ISCAS'03.*, vol. 5. IEEE, 2003, pp. V–V.
[30] L. Batina et al., "In hardware we trust: Gains and pains of hardware-assisted security," in *Proc. of the 56th Annu. Design Autom. Conf. 2019*, 2019, pp. 1–4.
[31] H. Ma et al., "Xmark: dynamic software watermarking using collatz conjecture," *IEEE Transac. on Inform. Forens. and Secur.*, vol. 14, no. 11, pp. 2859–2874, 2019.
[32] C. Iwendi et al., "Keysplitwatermark: Zero watermarking algorithm for software protection against cyber-attacks," *IEEE Access*, vol. 8, pp. 72 650–72 660, 2020.
[33] S. Mahdavifar et al., "Effective and efficient hybrid android malware classification using pseudo-label stacked auto-encoder," *Journ. of network and sys. manage.*, vol. 30, pp. 1–34, 2022.
[34] Q. Hao et al., "A hardware security-monitoring architecture based on data integrity and control flow integrity for embedded systems," *Applied Sciences*, vol. 12, no. 15, p. 7750, 2022.
[35] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Comput. Surv. (CSUR)*, vol. 54, no. 3, pp. 1–38, 2021.
[36] P. Boufounos and S. Rane, "Secure binary embeddings for privacy preserving nearest neighbors," in *2011 IEEE Intern. Works. on Inform. Forens. and Secur.*, 2011, pp. 1–6.
[37] M. Borowski et al., "Anomaly behaviour tracing of cheri-risc v using hardware-software co-design," in *2023 21st IEEE Interreg. NEWCAS Conf. (NEWCAS)*. IEEE, 2023, pp. 1–5.
[38] S. P. Kadiyala et al., "Lambda: Lightweight assessment of malware for embedded architectures," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 4, pp. 1–31, 2020.
[39] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?" in *Proceedings of the 2018 on Asia conference on computer and communications security*, 2018, pp. 457–468.

[40] M. S. Escobar et al., "Combined generative topographic mapping and graph theory unsupervised approach for nonlinear fault identification," *AIChE Journal*, vol. 61, no. 5, pp. 1559–1571, 2015.
[41] M. S. Escobar, H. Kaneko, and K. Funatsu, "On generative topographic mapping and graph theory combined approach for unsupervised non-linear data visualization and fault identification," *Computers & Chemical Engineering*, vol. 98, pp. 113–127, 2017.
[42] C. Ding, S. Sun, and J. Zhao, "Mst-gat: A multimodal spatial–temporal graph attention network for time series anomaly detection," *Information Fusion*, vol. 89, pp. 527–536, 2023.
[43] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 20–38.
[44] F. Javed, M. K. Afzal, M. Sharif, and B.-S. Kim, "Internet of things (iot) operating systems support, networking technologies, applications, and challenges: A comparative review," *IEEE Communi. Surv. & Tutorials*, vol. 20, no. 3, pp. 2062–2100, 2018.
[45] H. Wang et al., "Enabling micro ai for securing edge devices at hardware level," *IEEE Journ. on Emerg. and Selected Topics in Circ. and Sys.*, vol. 11, no. 4, pp. 803–815, 2021.
[46] X. Iturbe, B. Venu, J. Jagst, E. Ozer, P. Harrod, C. Turner, and J. Penton, "Addressing functional safety challenges in autonomous vehicles with the arm tcl s architecture," *IEEE Design & Test*, vol. 35, no. 3, pp. 7–14, 2018.
[47] Y. Gao et al., "Malware detection by control-flow graph level representation learning with graph isomorphism network," *IEEE Access*, vol. 10, pp. 111 830–111 841, 2022.
[48] G. et al., "Optiwise: Combining sampling and instrumentation for granular cpi analysis," in *2024 IEEE/ACM International Symp. on Code Generation and Optimization (CGO)*. IEEE, 2024, pp. 373–385.
[49] P. Krishnamurthy et al., "Anomaly detection in real-time multi-threaded processes using hardware performance counters," *IEEE Trans. on Inform. Forensics and Security*, vol. 15, pp. 666–680, 2019.
[50] C. M. Bishop et al., "Gtm: The generative topographic mapping," *Neural comput.*, vol. 10, no. 1, pp. 215–234, 1998.
[51] T. Kohonen and T. Kohonen, "Learning vector quantization," *Self-organizing maps*, pp. 175–189, 1995.
[52] S. Yang et al., "Robust and efficient star identification algorithm based on 1-d convolutional neural network," *IEEE Transac. on Aeros. and Electronic. Sys.*, vol. 58, no. 5, pp. 4156–4167, 2022.
[53] P. Veličković et al., "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.
[54] H. Gao et al., "Tsmae: a novel anomaly detection approach for internet of things time series data using memory-augmented autoencoder," *IEEE Transac. on network scien. and eng.*, 2022.
[55] A. Siffer et al., "Anomaly detection in streams with extreme value theory," in *Proc. of the 23rd ACM SIGKDD intern. conf. on knowledge disco. and data min.*, 2017, pp. 1067–1075.
[56] E. M. B. Consortium *et al.*, "The embedded microprocessor benchmark consortum," 2008.
[57] B. et al., "Verified security for the morello capability-enhanced prototype arm architecture," in *European Symposium on Programming*. Springer International Publishing Cham, 2022.
[58] R. Grisenthwaite et al., "The arm morello evaluation platform—validating cheri-based security in a high-performance system," *IEEE Micro*, vol. 43, no. 3, pp. 50–57, 2023.

Chandrajit Pal is currently associated with the Embedded and Intelligent Systems (EIS) Research Group, University of Essex, UK as a Senior Research Officer. Prior to that, he worked as a National post-doctoral fellow at IIT Hyderabad, India and as an AI Research Engineer at Ceremorphic Inc. His research interests mainly include computer vision and signal processing algorithms, custom computing using FPGAs, embedded systems and HW/SW co-design.

Sangeet Saha is currently associated with the Embedded and Intelligent Systems (EIS) Research Group, University of Essex, UK as a Lecturer. Prior to that, he worked as a lecturer at the University of Huddersfield, UK, and Senior research officer (Postdoctoral scholar) at the University of Essex, UK. His current research interests include real-time scheduling, scheduling for reconfigurable computers, real-time and fault-tolerant embedded systems, and cloud computing. He published several of his research contributions in conferences like CODES+ISSS, ISCAS, Euromicro DSD, and in journals like ACM TECS, IEEE TCAD, IEEE TSMC.

Xiaojun Zhai (SM'21) is currently a Reader in the Embedded Intelligent Systems Laboratory at the University of Essex. He has authored/co-authored over 140 scientific papers in international journals and conference proceedings. His research interests mainly include the design and implementation of digital image and signal processing algorithms, custom computing using FPGAs, embedded systems and hardware/software co-design.

Gareth Howells (Senior Member, IEEE) is currently a Professor at the School of Computer Science and Electronic Engineering, University of Essex, U.K., and the Founder, Director, and Chief Technology Officer of Metrarc Ltd., a university spin-out company. He has been involved in research relating to pattern recognition and image processing for over 30 years and has published over 200 articles in the technical literature, co-editing two books, and contributing to several other edited publications. His core research interests include applying soft computing and pattern recognition techniques to the domains of device authentication, biometrics, secure communications, and identity management.

Klaus D. McDonald-Maier is currently the head of the Embedded and Intelligent Systems Laboratory and director of research at the University of Essex, Colchester, U.K. He is also the founder of UltraSoC Technologies Ltd., the CEO of Metrarc Ltd., and a visiting professor at the University of Kent. His current research interests include embedded systems and SoC design, security, development support and technology, parallel and energy-efficient architectures, computer vision, data analytics, and the application of soft computing and image processing techniques for real-world problems. He is a member of VDE and a fellow of the BCS and IET.