NIRVANA: Non-Invasive Real-time VulnerAbility ANAlysis for RISC-V Processor

Jiacheng Zhu, Xuqi Zhu, Michal Borowski, Huaizhi Zhang, Chandrajit Pal, Sangeet Saha,
Dongbing Gu, Klaus McDonald-Maier and Xiaojun Zhai
University of Essex, Colchester, United Kingdom
{jz23222, xz18173, mb19424, hz24245 chandrajit.pal, sangeet.saha, dgu, kdm, xzhai}@essex.ac.uk

***Abstract*—Embedded systems are increasingly susceptible to attack from malicious software, posing a significant threat to critical infrastructure and data. Various shreds of evidence reveal the unknown nature of attacks. In this manuscript, we propose a novel abnormal behaviour monitoring and detection system by designing a self-supervised hardware-based Self-Organizing Map (SOM) algorithm which continuously monitors the execution status of an embedded program and the behaviour of the entire platform as a whole. Our design boasts low resource utilization, high speed, effectiveness, and broad compatibility, making it suitable for real-time detection of malicious behaviour in resource-constrained embedded systems. Experimental trials were conducted on the Piccolo RISC-V processor being prototyped on an FPGA, which achieved an impressive 96% accuracy in detecting malicious programs, at the cost of a marginal 10% increase in resource consumption in comparison to its vanilla counterpart.**

***Index Terms*—Embedded system security, abnormal behaviour detection, feature extract, self-organising map (SOM), Continuous Collection Module (CCM).**

I. INTRODUCTION

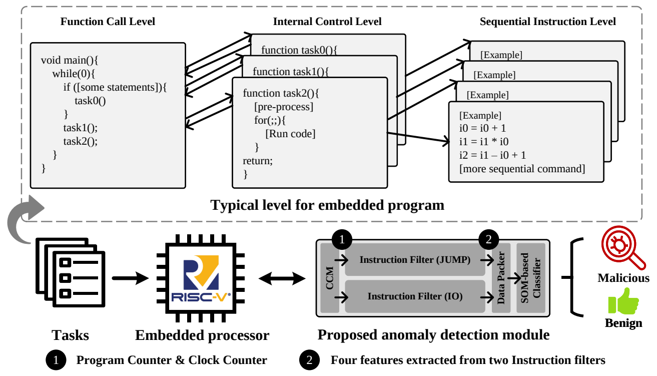
Embedded systems play a pivotal role across a wide spectrum of applications, spanning control systems, data management, and transmission to name a few. Their significance comes from the ability to execute real-time control tasks in various devices, handle critical tasks including data acquisition and process information efficiently with reduced power consumption and cost. Embedded systems typically consist of a microcontroller, along with integrated peripherals and associated software in a single device. The microcontroller primarily receives data from the sensor and drives another controller by changing the status of the I/O port in the device. Benefiting from its formidable capabilities it finds widespread use in various sectors like automotive, finance, aerospace, and healthcare, where personal safety is paramount [1]. However, with the rise of IoT devices, embedded systems face an increasing number of security threats [2].

In spite of various attempts security remains a primary concern for embedded systems [3], [4]. Despite their traits of low power and high energy efficiency, the complexity of modern embedded systems is still increasing with numerous powerful peripherals like DMA and CAN driver [5], [6]. Currently, to ensure data reliability, security systems often apply for comprehensive protection throughout the whole data chain including receiving, processing and transmission, leading to increased power consumption and shorter battery life, especially in battery-operated systems. Moreover, the simple structure of most embedded devices leaves the core processor vulnerable in physical structure to various attacks, such as tampering and side-channel attacks, resulting in unpredictable behaviours. Furthermore, unlike desktop processors equipped with memory management units and Address Space Layout Randomization (ASLR), embedded systems always operate with fixed memory addresses for each instruction, increasing the risk of program tampering [7], [8].

To address these challenges, Zhai et al. [9] proposed to detect abnormal behaviour by tracking the Cycles per instruction (CPI) during program execution, as different CPI characteristics emerge during abnormal program execution with Self-Organizing Map (SOM) based feature classifier achieving over 98.4% anomaly detection accuracy implemented on Keil MCBSTM32F200. Muhamed Fauzi Bin Abbas et al. [10] designed a Hardware-Performance-Counter (HPC) based runtime anomaly detection software leveraging Support Vector Machine classifiers. Similarly, Bourdon et al [11], introduced another HPC-based anomaly detection methodology for monitoring smart industry devices, eliminating the need for modelling onboard software applications. The authors in [12] proposed SEQ-TSD, a Sequential Time Series-based detection framework for identifying Malware on unconstrained devices using a single HPC. Simultaneously, the paper [13] proposed a framework CARE, which enables HPC-based malware detection models resilient to resource competition among programs.

These proposed solutions have shown the huge potential of classifier-based program monitoring algorithms at the software level. However, the majority of these software-based solutions rely on a debug trace interface that is constrained by bandwidth limitations, especially when transmitting HPC data as additional debugging information. The software-based analysis algorithms also restrict the hardware processing speed, presenting a significant challenge in achieving high-speed real-time anomaly detection in embedded systems.

Based on the architecture of the software-level monitoring system for CPI analysis and software SOM [9], we proposed an abnormal behaviour detection system by employing a dedicated feature extractor together with an optimal hardware design of SOM classifier module, which enables the proposed solution to provide a real-time anomaly detection capability in embedded systems with RISC-V processor. The primary contributions of this paper are summarised as follows:


Function Call Level
Internal Control Level
Sequential Instruction Level
Typical level for embedded program
Tasks
Embedded processor
RISC-V
CCM
Instruction Filter (JUMP)
Instruction Filter (IO)
Data Packer
SOM-based Classifier
Malicious
Benign
Proposed anomaly detection module
1 Program Counter & Clock Counter
2 Four features extracted from two Instruction filters

```
void main(){
    while(0){
        if ([some statements]){
            task0()
        }
        task1();
        task2();
    }
}
```

```
function task0(){
function task1(){
function task2(){
    [pre-process]
    for(;;){
        [Run code]
    }
return;
}
```

```
[Example]
[Example]
[Example]
[Example]
i0 = i0 + 1
i1 = i1 * i0
i2 = i1 – i0 + 1
[more sequential command]
```

Fig. 1: Overall Workflow for abnormal behavior detection

- We proposed a feature extractor module based on the behavioural characteristics of the RISC-V processor on an embedded system that eliminates the need for HPC information to detect abnormal behaviours of the system while executing embedded programs by introducing real-time feature analysis in fundamental debug information. This workflow enables the proposed solution to become a non-invasive module for real-time abnormal detection on the embedded systems using a RISC-V processor.
- We streamline the computational requirements of the SOM algorithm, which reduces the resource utilization of feature classification on the hardware together with the feature extractor module providing a resource-efficient analysis in embedded systems.
- The propose a hardware-based abnormal behaviour detection system implemented in the ZCU104 evaluation board achieves a detection accuracy of 96%, requiring only 1/10 of the resources and 6% of the power consumption compared to the Piccolo RISC-V core itself.

II. Threat Model

A prevalent method for executing malicious code on an embedded system is through code injection attacks [1]. Attackers exploit vulnerabilities in exposed privileged communication interfaces or side channels to manipulate memory, thereby leading embedded processing to execute abnormal commands and access sensitive data. In these attacks, attackers typically modify some regions of the memory in the system, leading the processor to execute some commands out of the range of the program.

However, due to the simple hardware architecture of the embedded processor, embedded programs always have similar structures, thereby leading to predictable memory and behavioural patterns [14], [15]. This feature enables us to protect the embedded processor by identifying the unique features of the specific programs which can be regarded as the signature of the program. Any modification to the program will lead to changes in the features of the program, which can be used to protect it from tampering.

III. Proposed Methodology

Embedded programs typically follow a standardized structure, beginning with peripheral initialization. Following this, various algorithms are called by an infinite loop in the top level or triggered by interrupts. The function thus emerges as the fundamental scheduling unit to run the algorithm within the program's execution framework [15]. As illustrated in Fig. 1, a typical embedded program comprises three distinct levels of code logic:

- Function Call Level: This level corresponds to the main loop and the top function for each algorithm. It involves organizing the algorithm's structure around top-level function calls. Typically, high-level programs generate long-distance jumps when calling algorithm functions. This feature can be utilized for identifying switches among algorithms.
- Internal Control Level: Algorithm's internal control flow, including state machines and various loops within each sub-function. It organizes the detailed execution flow within functions.
- Sequential Instruction Level: This basic block of computation involves the sequential execution of code without any branch or jump instructions. It includes a significant number of computational instructions that are critical to the execution of the algorithm.

An embedded program has a long instruction sequence that includes many different types of instructions. It would require more hardware resources to get better processing performance when performing real-time analysis on all the instructions in sequence. To accurately capture the features of algorithms while significantly minimizing hardware resource utilization and performance requirements, we've chosen to focus on analyzing certain types of instructions to compress the original debug data. Two fundamental types of instructions are chosen as the analysis target, memory input/output (IO) operations and unconditional jumps, since they are always used frequently and exist in almost all programs.

Specifically, memory access instruction is a fundamental operation which provides the processor with the ability to access memory and peripherals. This operation is essential for embedded processors and is unavoidable in any program. In addition, an unconditional jump is also a fundamental operation that provides the ability to control flow statements that make the program execution immediately jump to another part of the code. Therefore, memory input/output (IO) operations and unconditional jumps carry critical information about the program execution, which can be used to analyse and identify program features.

In the following sections, we introduce the proposed abnormal behaviour detection system. The proposed system consists of three main components: Continuous Collection Module (CCM), Feature Extractor which includes two instruction filters, and SOM classifier as the schematic overview depicted in Fig .1. These components are responsible for extracting processor debug data, extracting features, and training the

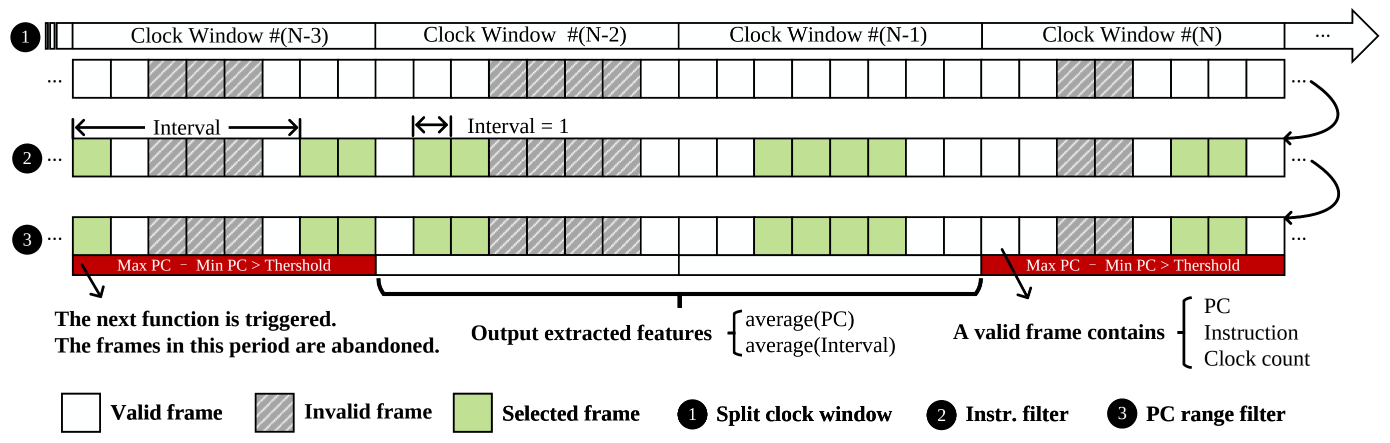


Fig. 2: Workflow for feature extractor

identification model, respectively. Data collected from the processor only comprises the most fundamental debugging information: the Program Counter (PC) and the current instruction. This minimalist approach ensures compatibility with various debugging interfaces. In this paper, RISC-V architecture is selected as the proposed target platform due to its clear ISA design.

A. Continuous Collection Module (CCM)

The CCM is an integral component connected to the debug port of the processor, designed to continually gather PC and current instruction values by decoding the various types of debug ports and transforming the debug data stream to the format of input of the feature extractor. An independent clock counter is also integrated within the CCM to provide the function of tracing the timeline of one executing program. This component will transmit the current instruction, PC, and clock count since reset to the next module in the form of a non-blocking data frame to be further processed in the feature extractor.

B. Feature Extractor

In an embedded platform, processors always run at a high frequency to improve performance, which would generate a dense debug information data stream that exceeds the classifier's processing capability. Therefore, it is necessary to use a feature extractor to distil the sparse and unique features of each algorithm from the real-time data stream. We proposed some steps to process the data stream as shown in Fig. 2. In this module, two features, address and invocation interval, are extracted from each instruction type. The address denotes the memory location of the currently executing instruction. Since embedded programs are rarely dynamic changes, the location of specific instructions contains important program features. The invocation interval indicates the time gap between two invocations of a particular instruction type, also contributing to the program's distinctive feature. A total of four features from the two instruction types will be extracted in this module. We'll then introduce the specific extraction process.

In the initial step, the data stream is divided into many distinct windows at fixed intervals determined based on the clock count in each frame. All valid data in each window are captured within the respective time span. Subsequently, frames of the specified instruction type (Memory IO and Jump) are then selected in each clock window. The next step is to calculate the PC range of the selected frames (i.e. PC range = maximum PC values - minimum PC values) in each clock window to identify the top-level long-distance jumps executed by the top-level scheduler which are the sign of the algorithm switch. If the PC range of selected frames exceeds the threshold (i.e. the new function starts processing), all frames in the corresponding clock window are discarded. Finally, the average PC values and the average time interval of two neighbouring selected frames in all retained clock windows are computed as features of this function. Each type of instruction has two features (i.e. the average PC and the average time interval), which are abstracted as a sample point in the four-dimensional hyperplane as shown in Fig. 3.

C. Self-Organized Map (SOM)-based classifier

Features extracted in the feature extractor module have transformed from dense debug data streams into sparse and unique features. Once all normal program features are extracted, the next step is to use SOM to learn these features and construct a model capable of rapid inference. The training of supervised neural networks like CNNs introduces higher parameter counts, extensive computational resources, prolonged training periods, and low inference efficiency, making it unsuitable for deployment on resource-constrained embedded devices. Machine learning algorithms such as SOM excel in classifying multidimensional data through similarity analysis, resulting in feature reduction. SOM has ultra-fast search speeds, making it the preferred choice for deployment on embedded devices.

SOM is an unsupervised artificial neural network that utilizes competitive learning strategies to acquire features of a dataset [16], [17]. It can segment large amounts of data based on the relationships among them, making it particularly

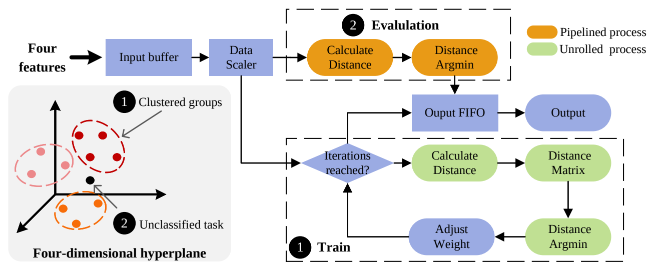


Fig. 3: Workflow for SOM-based Classifier

suitable for adaptive learning of program features with high data throughput. Compared to Convolutional Neural Networks (CNN), SOM has fewer parameters, and lower inference latency, suitable for running in embedded platforms with minimal degradation in accuracy. However, the algorithm prototype of SOM uses a lot of time-consuming computationally intensive operations such as square roots, significantly limiting its processing speed on the hardware.

In hardware design, to save training time and use fewer resources, various optimization methods are applied to the training process to eliminate most of the time-consuming operations. Fig.3 illustrates the process flow of the SOM-based Classifier at the High-Level Synthesis (HLS) stage. We applied UNROLL and PIPELINE optimizations to various processes in the SOM module. For operations requiring enumeration during training, all source data is synthesised into independent parallel processes to enable partial parallel optimization and acceleration, which is achieved by the unroll optimization in HLS. During inference, the calculations are typically divided into multiple steps, each taking several cycles to find the nearest group of the unclassified sample. By building a data pipeline, intermediate steps can also be concurrently calculated, thereby reducing input latency. Furthermore, some optimizations are carried out in several steps in SOM to reduce resource consumption and simplify the computation during the hardware implementation.

1) Training: The first step of training is initialising a set of neurons with uniform and dispersed weights, which are the random values with the same dimensions as the input dataset. Typically, the number of neurons must be greater than the expected total number of algorithms to be classified to provide some necessary redundancy. If too few point numbers are selected, it will degrade the robustness of the training. Otherwise, if the number of initialized points is too high, it will cost too much time and resources to train. In the hardware design of this module, the number of initialized weight points is twice the expected number of features to ensure a balance between resources and efficiency. Similarly, the number of iterations during the training process is also a parameter to consider. After experimenting, it was found that doubling the number of iterations of the training data leads to better results with a balance between time and accuracy.

During training, one random input element in the training dataset is first selected as the training point, and the distance is computed between this point and each neuron. In hardware design, the distance computation uses the Manhattan distance which is the sum of the absolute values of the differences in each dimension to replace the Euclidean distance to avoid the resource-consuming multiplication and square root operations.

$$D_j = \sum_{l=1}^{N} |K_l - w_{jl}| \tag{1}$$

$$D_{j,target} = argmin(D_j) \tag{2}$$

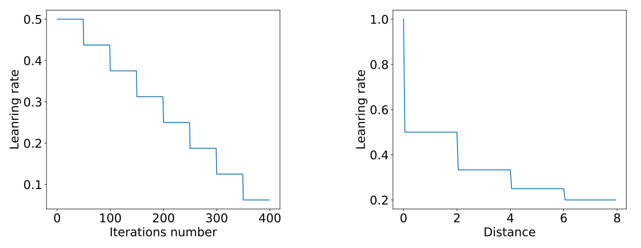
where D_j represents the distance of the j-th neuron relative to the given data, and w_{jl} represents the weight of the j-th neuron on the l-th dimension. Then we need to find the neuron with the lowest distance related to the given data point, denoted as $D_{j,target}$. After this step, it is necessary to compute the distances between all neurons, which can also be referred to as the distance Matrix, to apply subsequent weight diffusion. When calculating distances, the L1 norm is still used to save resources and time.

$$M_{ij} = \sum_{l=1}^{N} |w_{il} - w_{jl}| \tag{3}$$

M_{ij} represents the distance between the i-th and j-th neurons. After completing the distance-related calculations, the weights of the neurons can be updated accordingly.

$$w_{il} = w_{il} - \mu_0 \mu_1 (K_l - w_{il}) \tag{4}$$

μ_0 is a hyperparameter that decays over time, decreasing from a specified maximum value as training progresses. μ_1 is a distance decay parameter computed based on the distance matrix, used to update the weights of other points around the target neuron.



(a) learning rate decay for μ_0 (b) learning rate decay for μ_1

Fig. 4: Learning rate decay in training

To reduce the resources required when computing learning rate decay in hardware design, step-based decay functions are used for both learning rates. An exponential decay function is used for the decay process between neuron distances, which helps to avoid clustering among neurons. Compared to linear decay in each epoch of training, this approach can be optimized to store the value of the learning rate in the lookup table (LUT) to avoid division operations, which significantly reduces resource usage. Fig. 4 describes the trend of μ_0 and

μ_1 across 400 training epochs with dynamically decaying learning rates. In Fig. 4. (a), the learning rate for μ_0 follows an eight-segment linear decay pattern from 0.5 to 0 as training progresses. Fig. 4. (b) illustrates the learning rate μ_1, which reduces exponentially across four segments from 1 to 0 based on the distance between neurons. These attenuation approaches allow for achieving high accuracy while optimizing resource utilization. At this point, after repeating the above process several times, the training of all neurons is complete, and the neurons are ready to be used in inference.

2) Verification: Upon completion of model training, the neuron's weights can be efficiently utilized to perform high-speed classification inference by the enhancement of the pipelined process. Data inference involves executing only two steps of the training process: data distance calculation and identifying the nearest neuron by finding the minimum distance. Taking into account the FPGA's pipeline, the inference process typically requires 5 clock cycles and shares the resource used in training to execute the entire operation.

IV. EXPERIMENTATION AND RESULTS

The abnormal behaviour detection system and its target processor, Piccolo, have been prototyped on FPGA to evaluate its resource utilization and accuracy. CHERI Piccolo RISC-V processor is a high-performance embedded processor design with a Verifier Debug Interface to trace the execution [18]. ZCU104 is a ZYNQ Ultrascale+ FPGA platform with both general-purpose ARM processor (PS) and programmable logic (PL) used to prototype the proposed system, including processor and classifier.

A. Dataset Used

TABLE I: Details of Benchmark

Benchmarks	Description	ITERATION_COUNT
a2time	Angle to Time Conversion	400-1200
rspeed	Road Speed Calculation	5-15
bitmnp	Bit Manipulation	700-2100
idctrn	Inverse Discrete Cosine Transform	2-6
puwmod	Pulse Width Modulation	2-6
tblook	Table lookup and interpolation	25-105
ttsprk	Tooth to Spark	500-1500

Seven algorithms from widely recognized EEMBC's Auto-Bench 2.0 Benchmark are selected to train the SOM classifier [19]. As they have almost the same dataset and similar workflow that makes it harder to identify each. In order to simulate the randomness, their parameter is also randomly selected. Referring Table. I, ITERATION_COUNT is the key parameter that represents the number of times an algorithm is executed repeatedly which controls the algorithm's run time. In order to generate a continuous dataset, seven algorithms are re-ordered and mixed with different parameters to form a new program. Each algorithm is treated as a separate function call scheduled by the main loop.

In the evaluation, according to the settings of the compilation script, 1-4 algorithms with parameters are randomly selected in each round of compilation, and then executed in Piccolo processor. After several rounds of compilation and execution, a total of 257 debug data from seven different algorithms are generated following this workflow, and then processed by the feature extractor to generate the offline data set of the run-time debug data feature. By using simulations, we've achieved results that match the hardware implementation of SOM. Using these simulated results, a series of tests are performed to evaluate the classification accuracy. Also, the entire hardware design is verified on FPGA to gather the area and resource utilization.

B. System Implementation

During the evaluation, the metrics primarily include the correct recognition rate (true positives (T_p) and true negatives (T_n)), the rate of misclassified samples (false positives (F_p)), and the rate of samples incorrectly classified as unknown (false negatives (F_n)). Using these metrics, we can compute the accuracy, precision, and recall rates for the proposed system. The performance of classification is shown in Table II.

TABLE II: Performance results for the benchmarks

Benchmark	Accuracy (%)	Precision (%)	Recall (%)
a2time	94.88	90.45	85.49
rspeed	93.25	79.90	82.60
bitmnp	96.24	94.35	88.63
idctrn	95.47	85.85	88.00
tblook	96.33	63.04	86.56
ttsprk	95.81	76.41	77.14
puwmod	98.63	93.93	83.78
Average	95.80	83.42	84.60

Accuracy: This represents the proportion of correctly labelled samples, computed by the ratio of (T_p + T_n) to the total number of samples.

Precision: It indicates the proportion of positively labelled samples to the correct labels, measuring the classifier's resistance to malicious attacks. It's computed by $T_p/(T_p + F_p)$.

Recall: This signifies the proportion of samples that should have been positively labelled and were correctly labelled as such, measuring the classifier's resistance to false negatives. It's computed as $T_p/(T_p + F_n)$.

C. Hardware resource utilization

TABLE III: Resource utilization on ZCU104 at 100MHz

Component	LUT (utilization %)	FF (utilization %)
Piccolo	34411 (87.18%)	15013 (82.70%)
Feature Extractor	1474 (3.73%)	1172 (6.45%)
SOM	3567 (9.03%)	1768 (9.73%)
CCM	18 (0.04%)	200 (1.10%)
Total	39470 (100%)	18153 (100%)

Hardware validation for anomaly behaviour detection is prototyped on the ZCU104 Ultrascale+ Zynq validation board, consisting of both Programmable Logic (PL) and ARM A53 core (PS). This setup enables the concurrent validation of a random program generator and the system on its platform, facilitating uninterrupted data transmission and testing programs through the AXI interface. To modify the program

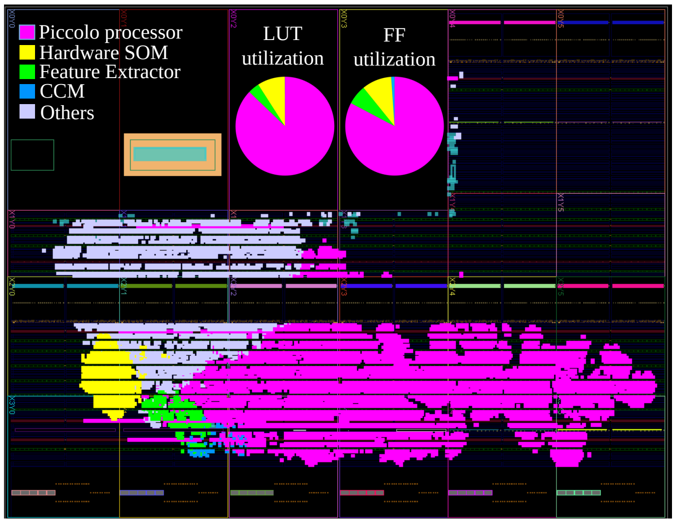


Fig. 5: Resource and area usage for each component

TABLE IV: Power Consumption on ZCU104 at 100MHz

Platform	Static Power	Dynamic Power
Piccolo	0.596W	0.481W
Piccolo & NIRVANA	0.596W	0.51W

executed by the Piccolo, an additional component is also integrated into the hardware. Fig. 5 and Table. III prove that abnormal behaviour detection systems designed for embedding onboard the proposed anomaly detection module (Feature Extractor, SOM and CCM modules) can efficiently execute a range of non-invasive detection tasks with only approx $12-16\%$ resource utilization of the entire system. Table IV illustrates that NIRVANA requires only a 6% (0.481W V.S. 0.51W) increase in dynamic power consumption to accurately perform feature extraction and classification tasks. Its exceptional compatibility demonstrates that the anomaly detection system can be inserted and integrated into most types of embedded processors, enabling effective monitoring and diagnosis of abnormal behaviour while incurring minimal resource overhead.

V. Conclusion

This paper presents a SOM-based anomalous behaviour detection system that can analyse instruction-related data of the Piccolo RISC-V processor in real time to protect the embedded system from the threat of malicious attacks. The system uses an optimised SOM algorithm for hardware implementation, ensuring efficient classification of program features. Only 10% of total resource overhead is introduced when integrated with the Piccolo processor. The entire system runs in real-time on the ZCU104 platform and achieves an impressive 96% detection accuracy.

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. 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] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi, "Security as a new dimension in embedded system design," *Proceedings - Design Automation Conference*, pp. 753–760, 2004. [Online]. Available: https://dl.acm.org/doi/10.1145/996566.996771

[2] 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, pp. 567–575, 11 2015.

[3] B. Yuce, P. Schaumont, and M. Witteman, "Fault attacks on secure embedded software: Threats, design, and evaluation," *Journal of Hardware and Systems Security*, vol. 2, pp. 111–130, 6 2018. [Online]. Available: https://link.springer.com/article/10.1007/s41635-018-0038-1

[4] A. I. Molcut, S. Lica, and I. Lie, "Cybersecurity for embedded systems: A review," *2022 15th International Symposium on Electronics and Telecommunications, ISETC 2022 - Conference Proceedings*, 2022.

[5] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 1295–1308, 12 2006.

[6] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," *2015 13th Annual Conference on Privacy, Security and Trust, PST 2015*, pp. 145–152, 8 2015.

[7] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 574–588, 2013.

[8] B. Jacob and T. Mudge, "Virtual memory: Issues of implementation," *Computer*, vol. 31, pp. 33–43, 6 1998.

[9] X. Zhai, K. Appiah, S. Ehsan, G. Howells, H. Hu, D. Gu, and K. D. McDonald-Maier, "A method for detecting abnormal program behavior on embedded devices," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 8, pp. 1692–1704, 2015.

[10] M. F. B. Abbas, S. P. Kadiyala, A. Prakash, T. Srikanthan, and Y. L. Aung, "Hardware performance counters based runtime anomaly detection using svm," in *2017 TRON Symposium (TRONSHOW)*. IEEE, 2017, pp. 1–9.

[11] M. Bourdon, P.-F. Gimenez, E. Alata, M. Kaaniche, V. Migliore, V. Nicomette, and Y. Laarouchi, "Hardware-performance-counters-based anomaly detection in massively deployed smart industrial devices," in *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, 2020, pp. 1–8.

[12] A. P. Kuruvila, S. Karmakar, and K. Basu, "Time series-based malware detection using hardware performance counters," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2021, pp. 102–112.

[13] Y. Hu, S. Liang, M. Li, T. Xue, B. Zhang, and Y. Wen, "Care: Enabling hardware performance counter based malware detection resilient to system resource competition," in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 2022, pp. 586–594.

[14] F. Vahid and T. Givargis, "Embedded system design: A unified hardware/software approach," 1999. [Online]. Available: http://www.cs.ucr.edu/ vahid

[15] M. J. Pont, *Embedded C*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[16] T. Kohonen, "Essentials of the self-organizing map," *Neural Networks*, vol. 37, pp. 52–65, 1 2013.

[17] ——, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, pp. 59–69, 1 1982. [Online]. Available: https://link.springer.com/article/10.1007/BF00337288

[18] C. Heinz, Y. Lavan, J. Hofmann, and A. Koch, "A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors," *2019 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2019*, 12 2019.

[19] E. AutoBench, "1.1 benchmark software," 2011.