Benchmark Tool for Detecting Anomalous Program Behaviour on Embedded Devices

Michal Borowski[1] Sangeet Saha[2] Xiaojun Zhai[3] Klaus McDonald-Maier[4]

School of Computer Science and Electronic Engineering
University of Essex
Colchester, UK
{[1]mb19424, [2]sangeet.saha, [3]xzhai, [4]kdm}@essex.ac.uk

***Abstract*—This paper presents an open-source benchmark tool for anomaly detection in program behaviour, using program counter (PC) and instruction type information. It is introducing anomalies in artificial way, allowing for fine-grained evaluation with adjustable sliding window sizes and preprocessing configuration. The usage of the benchmark, including demonstrated data collection, does not require any additional hardware other than a standard computer. The benchmark uses the output of llvm-objdump program to focus on non-library code which allows for rapid evaluation of various detection methods with different configurations. The proposed tool extracts features derived from processor's PC and instruction type information and then utilizes the features to identify abnormal behavior using 4 different anomaly detection algorithms. New detection methods can be easily incorporated into the benchmark, which provides a solid foundation for evaluating novel, previously unseen methods against methods we selected for our experiment.**

***Index Terms*—anomaly detection, machine learning, benchmark, program counter**

I. Introduction

From the early days of computing, security was not always the top priority when designing programming languages and computer systems. As computers became ingrained into the daily lives of the masses, this became a source of problems and increased demand for both: people who attempt to protect computer systems and people who tried to attack these.

Conventional approaches at securing computer systems include antivirus software, CPU modes and access policies (e.g. non-executable stack), software compartmentalization [1], [2], secure versions of programming language functions (e.g. C functions with "_s" suffix like sscanf_s), stack canaries, address space layout randomization [1].

However, these approaches mitigate attacks rather than address why such attacks are possible. Capability Hardware Enhanced RISC Instructions (CHERI) [3] security enhancement attempts to address it by ensuring that memory is used only for the purpose the programmer designed it. While CHERI offers groundbreaking security during the execution of a legitimate program, it does not offer any guarantee that the program being loaded is, in fact, legitimate. For this reason, mission-critical computer systems may be equipped with an additional security layer that may take into account the system's expected behavior (derived dynamically and/or statically) and continuously monitor program behavior, looking for a deviation from baseline. Optimally such a security system should be implemented as an internal (to minimize the risk of tampering) hardware module separated from the monitored system (to avoid being compromised when the system is compromised [4]). Hardware design, implementation, and evaluation all take time and effort. Before an anomaly detection method is implemented in hardware, it may be useful to test its effectiveness using a software model. The benchmark tool presented in this paper attempts to provide means to do that in a relatively simple way.

All detection methods presented in this work generate a baseline program profile in safe environment, and then compare it against behaviour exhibited by the system in potentially unsafe environment. It is assumed that anomalous behaviour is any kind of behaviour unseen during training. This approach implies that the detection system has to be trained before it can be used. Additionally, it must have full branch coverage during training, in other words enter every possible path (otherwise program parts unseen during training may be wrongly classified as anomalies as mentioned by Feng et al. [5]). On the other hand, this approach allows to detect previously unseen (zero-day) attacks [6].

The benchmark tool presented in this paper uses PC values and corresponding instruction types for program profiling, primarily because these were easily obtainable from the QEMU emulator. In the past, various other attributes were used for program profiling. Sekar et al. [7] used sequences of system calls and corresponding PC, respectively constituting edges and states of a finite-state automaton (FSA). Feng et al. [5] used the difference between non-matching return addresses present on stack (virtual path) at the entrance of neighbouring system calls, a method called VtPath. In [8] Lu and Lysecky used worst case (WTEC) and best case (BTEC) execution time of basic blocks. In their later work [6] this method was referred to as "lumped timing model" (due to large WTEC and BTEC difference stemming from process scheduling and system interrupts) and compared against improved versions

This work is supported by the UK Engineering and Physical Sciences Research Council through grants EP/V034111/1, EP/X015955/1 and EP/V000462/1.

that incorporated multi-range timing, hardware performance counters (instruction cache misses, data cache misses), and utilized one-class support vector machine (SVM). Arora et al. [9] used PC values and instruction types to verify whether the program follows the control flow graph (CFG) at function-to-function and basic-block-to-basic-block levels, additionally they pre-computed hashes of code sections and compared them with hashes computed during processor execution, which was done by Kanuparthi et al. [10] as well (in both cases resulting in a small performance overhead, $<= 2\%$). Abbas et al. [4] chose hardware performance counters (HPC) with the most stable values (instruction cache misses, data cache misses, branch prediction misses, number of call instructions) and used their total counts during complete program runs to train SVM classifier. Krishnamurthy et al. [11] periodically collected HPC readings from multi-threaded processor (at 1 kHz sampling rate) and used sliding window approach, where HPC values within windows of multiple temporal lengths are used to generate a vector of input features to SVM classifier. Yoon et al. [12] counted how many times each type of system call occurred during complete program runs, and used such obtained system call frequency distributions (SCFD) as an input to clustering algorithm which outputted multiple decision boundaries. Shu et al. [13] partitioned the program by function calls and recorded function call frequencies for each caller. In our previous work [14], PC values collected at clocks per instruction (CPI) peaks were supplied to a self-organising map (SOM) classifier.

Several detection-time choices can be distinguished among methods presented in previous studies. Some performed detection after the end of complete execution [4], [12], which simplifies the task of behaviour modelling but does not allow to detect intrusions shortly after they happen. Some studies performed detection upon calls/returns of system-calls and regular functions, as well as entrances/exits of basic blocks [5], [7]–[10], [13], or by using a sliding window approach where detection was performed following every N-number of events [15] or every time interval [11], in all those cases the detection is harder but allows to react to intrusions in reasonable time. In this work we use a sliding window approach.

In this work we contribute an open source benchmark tool, incorporating a semi-automated method of generating a dataset suitable for quantitative analysis using fine-grained data point labelling and evaluation. We further present the results of an experiment undertaken with the proposed tool, comparing novel anomaly detection algorithms against the direct sequence match method (ngrams). Additionally, we demonstrate a method of collecting low-level CPU information without the need to use dedicated hardware.

II. Benchmark Tool Overview

The proposed benchmark tool consists of the main comparison script responsible for training and evaluating algorithms (customizable by GUI, see Fig. 1, or a configuration file), and additional parsing/extraction scripts used to transform raw QEMU trace files into low volume CSV files containing PC values and instruction types, suitable to be used as input of evaluated algorithms. The benchmark tools was designed with consideration of challenges associated with program intrusion detection. It provides an option to use relative PC values, anticipating that the system may load a program at different address each time. The diversity of program behaviour and large volumes of collected data prolong the training time, which may take hours, days or even months [13]. By recognizing ".text" section address range of programs to focus on non-library code execution, we reduced collected data volumes 30 times, allowing to train and test more methods with various configurations in shorter time. Obtaining ground truth labels that indicate when exactly an anomaly occurs during program execution is a challenge, this benchmark introduces anomalies artificially, by directly modifying data points (e.g. PC values, instructions) collected from baseline program runs. This way it can obtain labels for each data point, allowing for fine-grained evaluation as opposed to coarse-grained evaluation where labels are assigned to whole program runs. Fig. 2 illustrates the workflow used to undertake our experiment, "III. Experimental Setup" section describes it in more detail, including the use of aforementioned scripts.

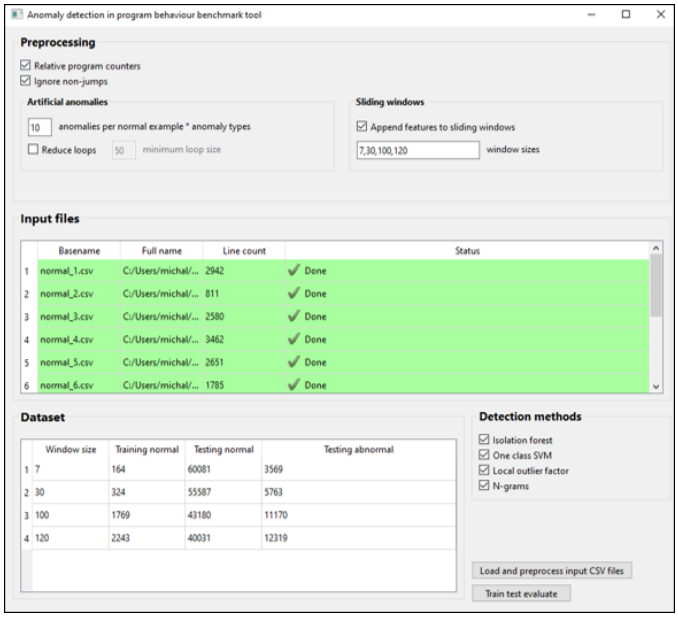

Fig. 1. Benchmark tool GUI.

III. Experimental Setup

A. Program choice and trace collection

The baseline program traces were collected by using qtrace utility (`qtrace -u exec ./program`) of QEMU emulator running CHERI-RISC-V [16]. The program "stack-mission.c", a CHERI adversarial mission called "Exploiting an uninitialized stack frame to manipulate control flow" [17]) was ran 10 times with varied user inputs, presented in Table I.

B. Trace parsing

1) Qtrace output: Despite simplicity of the stack-mission program, each collected trace log file contained around 74 megabytes of text, the trace of the stack-mission program being only a tiny part of it. A parsing script was used to extract PCs and instruction types (presented in Table II) from the relevant part (starting at the "main" function entry, and ending at "main" function return) of the trace log file.

TABLE I
USER INPUTS SUPPLIED TO THE STACK-MISSION PROGRAM, AND THEIR CORRESPONDING TRACE FILES.

stack-mission program keyboard input	Collected trace file name
==AA==AA==-=-AA====-	normal_1.log
=	normal_2.log
AA=-==-AAAA-=AA	normal_3.log
=-=—-AA=AA==AAAAAAAA	normal_4.log
–=AA==-AA-==AA-=	normal_5.log
AA-=AA=–	normal_6.log
AA=	normal_7.log
AAAAAA=	normal_8.log
-=-=	normal_9.log
AA–	normal_10.log

TABLE II
THE CONTENT OF NORMAL_1.CSV FILE

11FB2, addi
11FB4, sd
11FB6, sd
11FB8, addi
11FBA, mv
11FBC, sd
11FC0, sw
11FC4, auipc
11FC8, jalr
11FE8, addi
11FEA, sd
11FEC, sd
(... around 700 more lines)

2) Focus on ".text" section only: Only the trace from the execution of ".text" section was taken into account, effectively ignoring execution of library code (which may not be optimal when detection system is deployed in real world scenario because many attacks may target libraries [5], however this drastically decreases the number of extracted values, decreasing testing time and allowing to test relatively large number of heavy-computation models with different configurations [7]). In case of "normal_1.log" trace file, extracting PC and instructions from ".text" section only results in 2942 values, as opposed to 92047 values that would be extracted if ".text" section only restriction didn't apply.

```
{
    "total": [
        72352,
        74806
    ],
    (...)
    "main": [
        73650,
        73702
    ],
    "init_cookie_pointer": [
        73704,
        73794
    ],
    (...)
}
```

Fig. 3. Example part of JSON file containing function ranges from ".text" program section.

3) Workflow: After compilation of the stack-mission program, llvm-objdump was ran for it and the output was stored in a file containing disassembly of each section of stack-mission executable. A script was used to extract function ranges from that file (the beginning and ending addresses) from each function within ".text" section as well as the whole ".text" section itself. Resulting json file contains functions names as keys and their minimum and maximum PCs (ranges) as values (including one special "total" key of which range encapsulates the whole ".text" section).

Obtaining such json file allowed to extract desired values from the trace log files.

C. Dataset overview

Each of 10 baseline program runs used for training were copied 10 times, resulting in 100 copies. Then a single

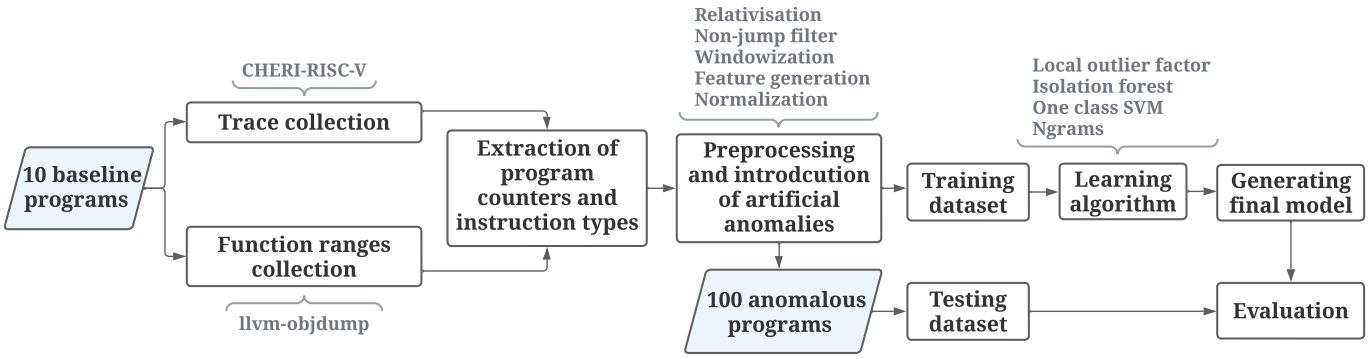

Fig. 2. Experimental setup overview.

anomaly was introduced in each copy by setting a random section (with variable length) of PC to random values within program range (see Fig. 4), and replacing instructions within the same region to other random instructions. Such artificial introduction of anomalies is not perfectly representing real world attacks, however it allows to generate ground truth labels and use fine-grained evaluation as a result.

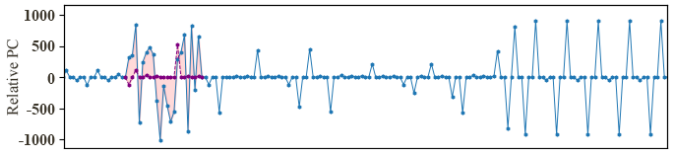

Fig. 4. A section of a program containing a single anomaly. Each dot represents a single relative PC value. Red area of the plot indicates discrepancy between the baseline PCs (collected with qtrace) and anomalous ones introduced artificially.

D. Pre-processing

Various pre-processing options are available through GUI or the configuration file. In this study the following options were used:

1) Relative program counters: A program can be loaded at different addresses, meaning that previously collected values may not match. For that reason the benchmark has an option to make PC values relative to their preceding value.

2) Non-jump instructions filter: PC changes equal to the size of a single instruction do not bring much value to the quality of program profiling nor anomaly detection. The benchmark has an option to remove such data points and leave only those that are a result of branching (e.g. due to function calls, returns and conditional statements like "if" or "switch").

3) Sliding window size: In configuration file we can specify a comma-separated list of window sizes that will be used to repetitively train and test detection models.

According to Sekar et al. [7] the number of ngrams grows exponentially with N (window size), however this seems not to be the case in our dataset as shown in Fig. 6. Lack of exponential growth in our case may be resulting from the simplicity of the chosen program and the fact that loops take large portion of its execution, both leading to low sequence variety.

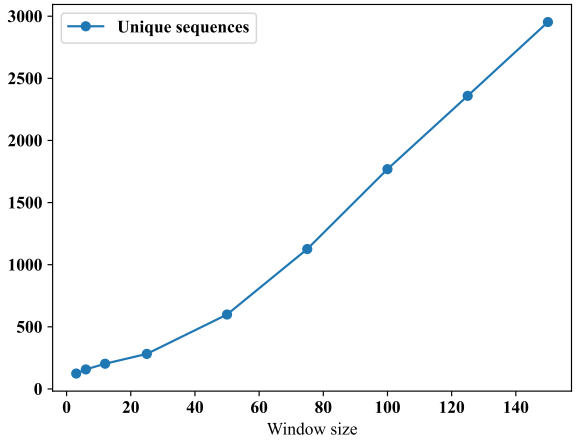

Fig. 6. The number of unique sequences (ngrams) found in the training dataset, in relation to window size.

4) Appending statistical features to sliding windows: Optionally we may append statistical features to each sliding window based on values they contain. Such features include: mean, standard deviation, the number of jumps, mean jump size.

5) Appending instruction IDs to sliding windows: After loading the dataset, the benchmark assigns an ID to each instruction type found in the dataset (e.g. addi=0, auipc=2, beq=3...). This way these instructions can be supplied to detection methods that expect numerical input.

6) Normalization: The following normalization formula was used:

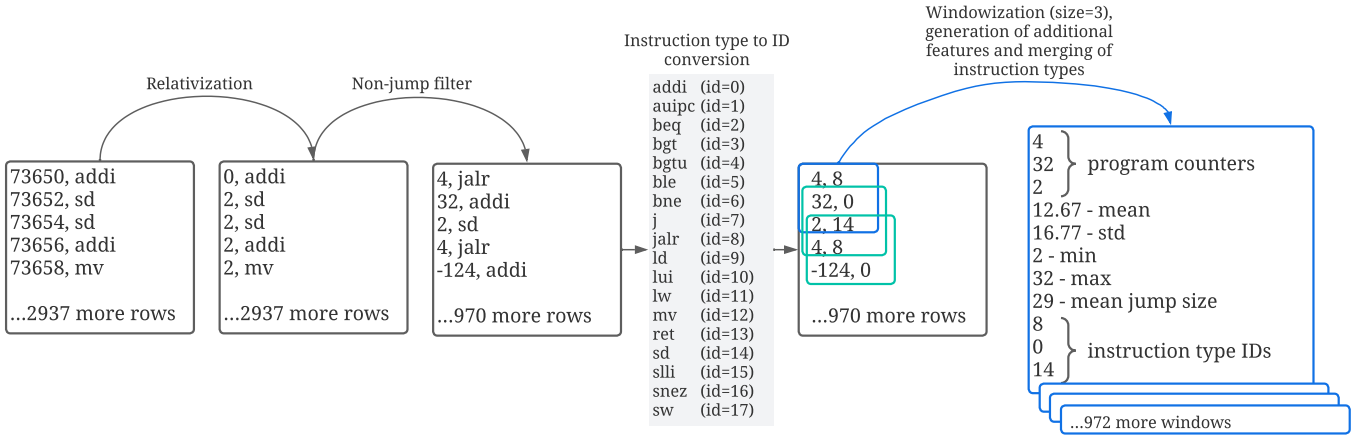

Fig. 5. Preprocessing and windowization illustrated using "normal_1.csv" file and window size of 3. This procedure is done for both: baseline dataset, and testing dataset. Data points surrounding jump instructions are retained during "non-jump filter", that is why some low relative PC values are still present despite applying the "non-jump filter".

$$x\_norm = (x - min\_val)/(max\_val - min\_val) \quad (1)$$

where minimum and maximum values were obtained from training dataset only.

E. Ground truth labels

A label is generated for every sliding window. The value of a label can be 0 (baseline), or 1 (anomalous). A window is labeled with 1 when it contains at least one artificially modified data point (PC and instruction type). Aside of label, each window is assigned a set of anomaly identifiers, which allows to keep track of anomalies that were detected.

F. Detection methods

We evaluated isolation forest [18], one class SVM [19], and local outlier factor [20], using their implementations from scikit-learn package. We compared them against a simpler but space expensive ngrams [15] method, where we stored all PC combinations that occurred during training with length N (similarly to FSA [7] method, which operated on system-calls and is equivalent to ngrams with $N = 2$).

IV. Evaluation

A. Main goals

From the perspective of an user who deployed a detection system in a computer system, two evaluation metrics may be particularly important. The first metric being the number of detected anomalies out of all anomalies that took place, and the other metric being the number of false alarms.

B. Characteristics

One way to evaluate the detection methods would be to compute confusion matrix based on all windows from the testing dataset, however that would not provide the first metric mentioned above. In order to provide the number of detected anomalies, each sliding window is assigned a set of anomaly IDs it covers, which means that a single anomaly may be a part of multiple consecutive windows. An anomaly is counted as detected when at least one sliding window with specific anomaly ID is classified as anomalous. Therefore the number of anomalies in a given program is absolute, in a sense that it is not affected by factors like sliding windows size. Unlike the number of anomalies, the number of false positives (sliding windows wrongly classified as anomalous), is affected by the sliding window size. That is because the benchmark considers a sliding window to be truly non-anomalous only when every PC it contains is non-anomalous (and the number of such windows is in fact dependent on window size). Multiple contiguous PC values are treated as a single anomaly.

C. Results

Figures 7-10 present the rates of detected anomalies and false positives using 4 different methods and 9 different sliding window sizes. Anomaly detection rate was calculated using the following formula:

$$x = detected\_anomalies/all\_anomalies \quad (2)$$

False positives rate was calculated using the following formula (where "nw" stands for normal windows):

$$x = nw\_classified\_as\_anomalous/nw \quad (3)$$

Ngrams was the best performing method, detecting all 100 out of 100 artificially introduced anomalies and having 0 false positives rate in all tested sliding window sizes.

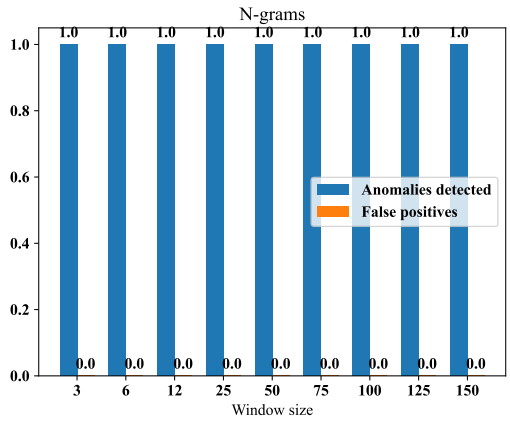

Fig. 7. Ngrams results.

From all other methods only the local outlier factor detected all anomalies without having any false positives (at window sizes 50 and 125), however at window sizes 3 and 6 it detected 0/100 and 1/100 anomalies respectively (which was most likely caused by the lower number of unique training windows when lower window size was used and the fact that "contamination" parameter was always set to fixed value of 0.001). These 2 methods did not result in any false positives at all regardless of window size.

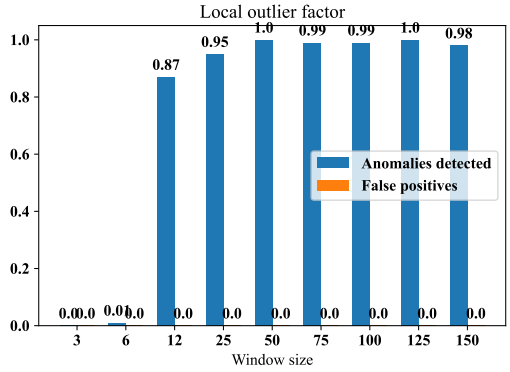

Fig. 8. Local outlier factor results.

One class SVM resulted in the highest false positives rate which together with detection rate appears to be negatively correlated with the sliding window size.

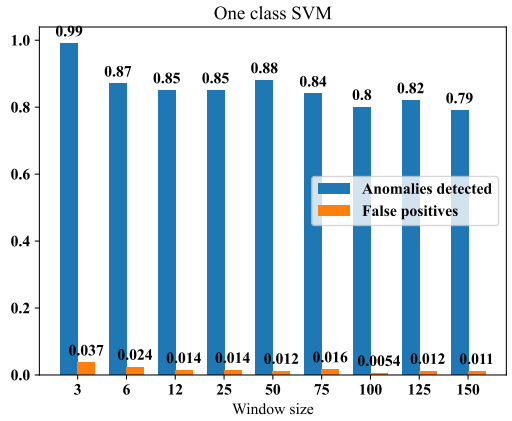

Fig. 9. One class svm results.

Isolation forest resulted in significantly lower false positives rate however its detection rate was fluctuating along different window sizes (with optimal range around 12-25).

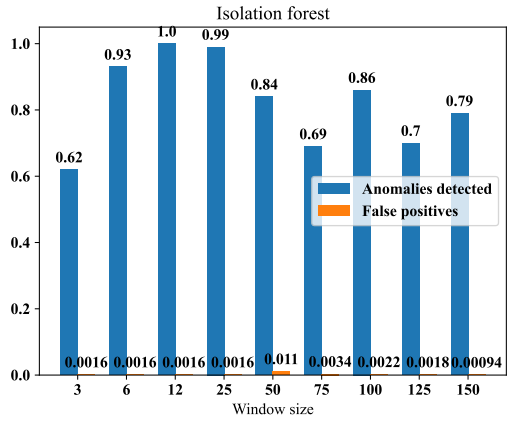

Fig. 10. Isolation forest results.

V. Conclusions

This paper presented an open-source benchmark tool for anomaly detection in program behaviour. The benchmark tool was used to compare performance of 4 different anomaly detection methods using fine-grained evaluation. High customizability of the benchmark allows to test different parameter settings of machine learning algorithms, apply different preprocessing operations (e.g. to decrease data volumes or simulate varying program load address) and test custom range of sliding window sizes. Additionally, a method of extracting program behaviour profiling information excluding library code was described, which does not require any dedicated hardware. The benchmark tool is available at [21], which includes source code, usage instructions and information about extending the tool to accommodate new anomaly detection methods. In the future, the benchmark tool may be extended by incorporating more program profiling features (e.g. timing information, hardware performance counters) and detection methods.

References

[1] H. M. Gisbert and I. Ripoll, "On the effectiveness of nx, ssp, renewssp, and aslr against stack buffer overflows." IEEE, 8 2014, pp. 145–152.

[2] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," vol. 2015-July. Institute of Electrical and Electronics Engineers Inc., 7 2015, pp. 20–37.

[3] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk." IEEE, 6 2014, pp. 457–468.

[4] M. F. B. Abbas, S. P. Kadiyala, A. Prakash, T. Srikanthan, and Y. L. Aung, "Hardware performance counters based runtime anomaly detection using svm," vol. 2017-January. IEEE, 12 2017, pp. 1–9, test.

[5] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," vol. 2003-January. IEEE Comput. Soc, 2003, pp. 62–75, 2003.

[6] S. Lu and R. Lysecky, "Data-driven anomaly detection with timing features for embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 24, pp. 1–27, 6 2019.

[7] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors." IEEE Comput. Soc, 2001, pp. 144–155.

[8] S. Lu and R. Lysecky, "Time and sequence integrated runtime anomaly detection for embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 17, pp. 1–27, 4 2017.

[9] 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.

[10] A. K. Kanuparthi, R. Karri, G. Ormazabal, and S. K. Addepalli, "A high-performance, low-overhead microarchitecture for secure program execution." IEEE, 9 2012, pp. 102–107.

[11] P. Krishnamurthy, R. Karri, and F. Khorrami, "Anomaly detection in real-time multi-threaded processes using hardware performance counters," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 666–680, 2019.

[12] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha, "Learning execution contexts from system call distribution for anomaly detection in smart embedded system." ACM, 4 2017, pp. 191–196.

[13] X. Shu, D. Yao, and N. Ramakrishnan, "Unearthing stealthy program attacks buried in extremely long execution paths," vol. 2015-October. Association for Computing Machinery, 10 2015, pp. 401–413.

[14] 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, pp. 1692–1704, 8 2015.

[15] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, pp. 151–180, 7 1998.

[16] "Cheri-risc-v qemu," 2022. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-qemu.html

[17] R. N. M. Watson, B. Davis, W. Filardo, J. Clarke, and J. Baldwin, "Cheri exercises: Exploiting an uninitialized stack frame to manipulate control flow." [Online]. Available: https://ctsrd-cheri.github.io/cheri-exercises/missions/uninitialized-stack-frame-control-flow/index.html

[18] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest." IEEE, 12 2008, pp. 413–422.

[19] B. Schölkopf, R. C. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt, "Support vector method for novelty detection," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12. MIT Press, 1999.

[20] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: identifying density-based local outliers," *ACM SIGMOD Record*, vol. 29, pp. 93–104, 6 2000.

[21] M. Borowski, X. Zhai, S. Saha, and L. Poli, "Benchmark tool." [Online]. Available: https://github.com/michalmonday/Program-Behaviour-Anomaly-Detection-Benchmark