Anomaly Behaviour tracing of CHERI-RISC V using Hardware-Software Co-design

Michal Borowski, Chandrajit Pal*, Sangeet Saha, Ludovico Poli, Xiaojun Zhai and Klaus D McDonald-Maier
University of Essex, Colchester, United Kingdom, ∗ Indian Institute of Technology, Hyderabad, India.
{mbl9424, sangeet.saha, lp17239, xzhai, kdm}@essex.ac.uk
∗ chandrajit.pal@ee.iith.ac.in

***Abstract*—Capability Hardware Enhanced RISC Instructions (CHERI) is an extension of conventional ISAs with capabilities enabling fine-grained memory protection. Recently, RISC-V ISA has been extended to CHERI-RISC-V (aka Flute) with additional support for CHERI. In this paper, we have proposed a lightweight continuous monitoring system (CMS) based on hardware-software co-design that communicates with the RISC -V to identify any abnormalities in its operational behaviour. The digital hardware of the functionality of CMS and the CHERI Flute RISC-V has been prototyped in the FPGA. The CMS extracts the different features from RISC-V and transmits them to the processing system via an API. Further, an anomaly detection program is being executed by the ARM processor residing in the PS portion of the ZYNQ. This program enables continuous evaluation of the system operation to spot hardware failure or unusual system behaviour. Finally, the complete design has been prototyped and verified on Zynq FPGA ZC706.**

Keywords - Capability hardware-enhanced RISC instructions (CHERI), Continuous monitoring system (CMS), Flute, RISC-V, PYNQ

I. Introduction

Anomalous program behaviour detection on RISC-V-based embedded devices is found in various studies. Authors in [1] conducted a thorough investigation to detect a fault using a counter-based built-in self-test strategy in a Rocket RISC-V microprocessor prototyped on FPGA. This study [2] created a dataset of execution traces containing Return Oriented Programming (ROP) exploitation on the RISC-V Instruction Set Architecture and used deep learning AI models like long short-term memory (LSTM) to distinguish exploited traces from non-exploited traces to detect ROP attacks. The authors in [3] proposed a methodology to perform real-time monitoring of software that kept track of hardware performance counters executing on embedded processors in cyber-physical systems. The time series data from the hardware performance counter measurements over a time window under well-known operating conditions are used to train a machine learning classifier.

Apart from RISC-V-specific mechanisms, the research has delved into developing security mechanisms for general embedded systems with multiple processors, and some of these techniques could be well applied in the case of RISC-V. Authors in [4] utilised the subcomponent timing information of software execution with features including intrinsic software execution, instruction cache misses, and data cache misses for anomaly detection. Zhai et al. [5] used a self-organizing map (SOM)-based approach by detecting abnormal program behaviour by proposing a methodology that extracts features derived from the processor's program counter (PC) and cycles per instruction, followed by utilising the features to identify abnormal behaviour using the SOM.

Recently, Capability Hardware Enhanced RISC Instructions (CHERI) is an extension of conventional Instruction Set Architectures (ISA) with capabilities enabling fine-grained memory protection and scalable software compartmentalization that has been popularised as a new way forward to enhance security in RISC-V. While RISC-V ISA has been extended to CHERI-RISC-V with more support for CHERI, Flute is an open-source 64-bit RISC-V processor with a five-stage, in-order pipeline. Designers in UoC [6] have extended the open-source Bluespec SystemVerilog (BSV) RISC-V core Flute (64-bit, 5-stage) with support for CHERI-RISC-V on the FPGA boards, and these implementations are still highly experimental to bring them into practical usage. This necessitates implementing and prototyping a complete end-to-end implementation of the CHERI-Flute-RISC-V processor on FPGA.

However, the employed tools can introduce malicious modifications into the designed system. While CHERI offers security during the execution of a legitimate program, it does not offer any guarantee that the program being loaded is, in fact, legitimate. Thus, computer systems may be equipped with an additional security layer that may take into account the system's expected behaviour (derived dynamically and/or statically) and continuously monitor program behaviour, looking for a deviation from the baseline. To achieve this, the processor's behaviour may be monitored in real-time through the extraction of featured data from its PC, instructions, and registers.

To this end, in this paper, we have proposed a lightweight hardware-software co-design-based continuous monitoring system (CMS) that interacts with the RISC-V in order to detect any anomalies in its operating behaviour. The CHERI Flute RISC-V and the CMS are implemented within the FPGA fabric. Due to the inherently parallel nature of the hardware execution, the RISC-V and CMS can be adapted for faster execution. This is a major advantage of designing efficient

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

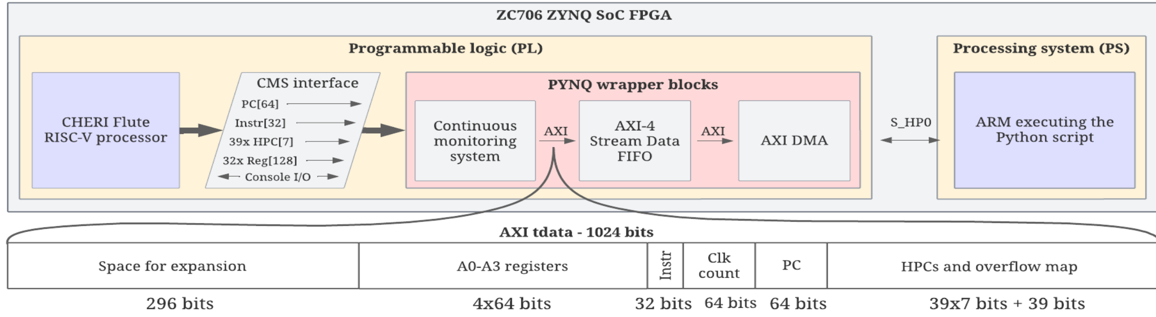

Fig. 1. Project structure and data transfer path.

monitoring hardware architectures able to fetch/extract data at higher data rates. However, similar to ASIC development, designing FPGA implementations for algorithmic analysis of the extracted information is more complex in comparison to software development. Hence these features are sent to the processing system through a developed API where the ARM microcontroller is executing an anomaly detection program that allows evaluating the system operation continuously to identify hardware failure or unusual system behaviour. Thereby, making a holistic end-to-end hardware-software co-design methodology initialising with the processor's behavioural data extraction to its anomaly detection.

The main technical contributions of this paper are summarised as follows:

- A complete end-to-end design initiating with the CHERI-FLUTE-RISC-V processor followed by designing our proposed continuous monitoring system, FIFO and a direct memory access module, as hardware modules have been prototyped on the programmable logic of FPGA surrounded by a PYNQ wrapper, to increase the throughput of the data required to be passed to the Processing system performing the anomaly detection in software owing to its sequential nature running on ARM microcontroller.
- The designed PYNQ wrapper utilises the continuous hardware monitoring module to filter and extract the trace information of the CHERI-Flute processor, including program counter, instruction, performance counters, general purpose registers and timing information, then it utilizes the features to verify the correctness of the program behaviour. These features are sent to the processing system through a developed Application processing interface (API) where the ARM microcontroller is executing an anomaly detection program that allows evaluating the system operation continuously to identify hardware failure or unusual system behaviour. Thereby, keeping a provision with the ability to debug the specification at run-time in future.

II. Proposed Methodology

A. Proposed PYNQ wrapper design

A typical PYNQ design includes a hardware design that configures the PL and a Python script running into the Linux-based PS on an ARM microcontroller, where the Python script is responsible for loading and interacting with the hardware design in PL (using PYNQ API) as shown in figure 1.

We designed the PYNQ wrapper for the open-source RISC-V processor (Flute) accompanying peripherals for loading a program into memory, console I/O, as well as storage, filtering and preprocessing of the collected trace data from the Flute processor etc. The hardware IPs within the PYNQ wrapper blocks and the accompanying Python scripts using PYNQ API interacts with the PL to collect the processor behavioural data and subsequently process the collected data. The hardware blocks residing within the PYNQ wrapper are directly connected to the Flute RISC-V processor as shown in figure 1. Data including program counter (PC), instruction, performance event indicating vector, general purpose registers and signals for console I/O are fetched from the Flute processor to the PYNQ wrapper blocks which are then transferred to the PS.

The PYNQ wrapper utilises our designed continuous monitoring hardware module to filter and extract information commonly used for anomaly detection in program behaviour from the program counter [5], [7], 39 hardware performance counters (HPCs) [3], [8], [9], time since last extracted item [10], [11], the corresponding instruction and 4 general purpose registers (A0 - A3) responsible for storing function parameters and return values (in case of A0 and A1). Initial filtering is done to reduce the amount of extracted data and collect only the data when a branch, jump or return instruction is executed (including any instruction that immediately follows these). Studies that use HPCs are often limited to collecting only a few of them in real-time, between 2 and 6, or collecting them with delays after multiplexing due to hardware limitations [3], [8], [12]. In this work, we overcome this and extract 39 HPCs by modifying the Bluespec source code of the Flute processor to propagate performance event-indicating signals into our proposed CMS module. We tested the PYNQ wrapper design by obtaining a baseline program profile and then comparing it

Fig. 4. Data collection and anomaly detection process.

against data collected from anomalous program run, the testing process was illustrated in figure 4 and explained in section III.

B. Hardware Performance Counter selection

The version of the RISC-V Flute processor we used (RV64ACDFIMSUxCHERI architecture) contains 85 performance counters from which features are extracted. We collected all their values during the exploratory program run and made a list of 37 event types that had at least one non-zero value, which we decided to keep together with “trap” and “interrupt” events.

C. Data transfer path

Referring to figure 1 the CHERI Flute processor (implemented in Bluespec Verilog language) is modified to propagate relevant signals outside of it to the Continuous Monitoring System, which groups and filters them. Signals propagated outside of the processor include program counter, instruction, 39 performance event indicating bits (each indicating a different performance event currently taking place) and all 32 general purpose registers including their CHERI-related metadata (e.g. pointer boundaries, permissions, object type). The CMS module receives these signals as inputs and performs initial filtering. The CMS counts each performance event and the clock cycles as well since the last collected item. If the data item is to be collected, it turns all values into a single 1024-bit vector and transfers it to AXI4-Stream Data FIFO. As shown in figure 1 contents of that 1024-bit vector include:

- Program counter (64 bits)
- Instruction (32 bits)
- Clock ticks count since the last extracted item (64 bits)
- 39 performance counters (7 bits each) indicating how many of each event occurred since the last extracted item
- Performance counters overflow map (39 bits), indicating which counters have to be treated as the modulo of 128 values (due to going over their maximum value by 7 bits)
- 4 selected general purpose registers (A0-A3) without their CHERI-related metadata (256 bits in total)

All data is collected as the instruction and program counter are passed from stage 1 to stage 2 of the CPU pipeline. After delivering the data to FIFO, the AXI DMA may be requested to transfer the contents of it into previously allocated contiguous memory. This kind of allocation was accomplished by using allocate function from the PYNQ API, and requesting the transfer is done by using PYNQ API. Control over the AXI DMA module is done from a controller running in PS. The FIFO has the capacity to store 2048 elements. A single run of the stack-mission program (described in the experimental setup section) results in around 1300 items being collected (under the assumption that it immediately receives data through standard input instead of waiting for it). After running the program, the Python script may request the data to be delivered through AXI DMA (S_HP0, high-performance connection) into a contiguous memory array allocated on PS.

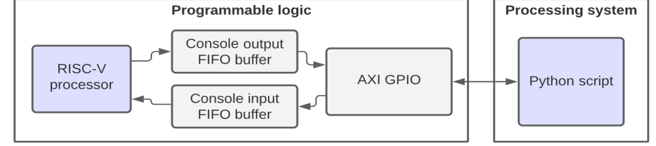

Fig. 2. Flute controlling console I/O

As shown in figure 2 we designed 2 FIFO buffers inside the PYNQ wrapper for controlling the console I/O. One to store output characters before these are read, and one to store input characters. Reading and writing into these buffers is accomplished by using an AXI GPIO module together with signal edge detectors ensuring reading/writing is done efficiently for no longer than a single clock cycle.

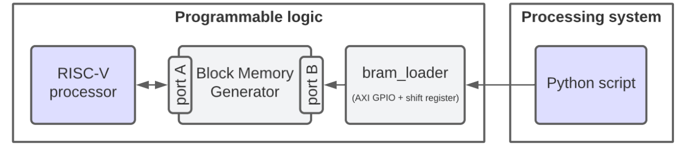

Fig. 3. Flute interacting with memory

Inside the PYNQ wrapper, we designed a Block Memory Generator that has 2 ports for reading and writing (figure 3). Port A - is connected to the Flute processor using signals to read/write data from/to memory and port B - is controllable from python, connected to PS through the bram_loader hierarchical block, allowing to load a program binary into memory.

III. EXPERIMENTAL SETUP OF THE DESIGN

A. Vulnerable program used in our test

To utilise the PYNQ wrapper, we used the vulnerable stack-mission program provided for the “Exploiting an uninitialized stack frame to manipulate control flow” CHERI-exercise [13], with modifications. The program is vulnerable because the buffer responsible for storing user input covers the same area of the stack as the function pointer variable in the function that is executed afterwards, so the uninitialized function pointer retains the previously stored value in the same memory location (ref. figure 5). Using a meticulously crafted input, a potential attacker may divert the program execution from ”**no_cookies**” function executed by default into the ”**success**” function of which address is supplied as a part of the input. We used the PYNQ wrapper to collect metrics from RISC-V processor running the stack-mission program and to detect the exploit, which was done using the N-grams method later on [14].

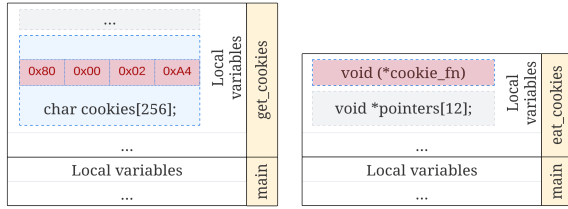

Fig. 5. Stack frames overlap, allowing to pre-set the uninitialized "**cookie_fn**" variable from "**eat_cookies**" function with the address of "success" function (0x800002A4).

B. How the test was done

First, we ran the stack-mission program 10 times with varying inputs to establish the baseline program behaviour profile. The input for training program runs was chosen specifically for the stack-mission program, attempting to contain a high variety of input *cookies* to cover all branches during the training phase. *Cookies* here signify a single-byte chunk received from the user input and are stored as a local variable in a buffer. After obtaining the baseline data, we ran the program with input that contained the address of the "success" function (0x800002A4) preceded by a sufficient number of equal signs ("="), each equal sign skipped 8 bytes in order to overwrite the exact position of the function pointer (ref. Table I). Our version of the stack-mission program distinguished 10 types of possible cookie types that result in slightly different behaviour of the program (due to custom implementation of the *isxdigit* function): 'AA', 'aA', 'Aa', '0A', 'A0', 'a0', '0a', '00', '-', '='. Failure to put a program into every possible valid state during online training may result in false positives while testing [15]. For that reason, we attempted to train our model with a large variety of inputs. We aggregated the Cartesian products of all 10 cookie types and produced a string containing 200 cookies where every cookie type was followed by every possible cookie type at least once. We split that string into 10 input strings with 20 cookies each.

TABLE I
USER INPUTS SUPPLIED TO THE STACK-MISSION PROGRAM

User input	Dataset
AAAAAAaAAAAaAA0AAAA0AAa0AA0aAA00AA-AA=	Training
aAAAaAaAaAAaaA0AaAA0aAa0aA0aaA00aA-aA=	Training
AaAAAaaAAaAaAa0AAaA0Aaa0Aa0aAa00Aa-Aa=	Training
0AAA0AaA0AAa0A0A0AA00Aa00A0a0A000A-0A=	Training
A0AAA0aAA0AaA00AA0A0A0a0A00aA000A0-A0=	Training
a0AAa0aAa0Aaa00Aa0A0a0a0a00aa000a0-a0=	Training
0aAA0aaA0aAa0a0A0aA00aa00a0a0a000a-0a=	Training
00AA00aA00Aa000A00A000a0000a000000-00=	Training
-AA-aA-Aa-0A-A0-a0-0a-00---=	Training
=AA=aA=Aa=0A=A0=a0=0a=00=-==	Training
====================**A4020080**00000000	Testing

We used a sliding window to compute unique sequences of PC collected during baseline program runs. Collected sequences (N-grams with N=10) were used as a lookup for the anomalous program run. If a specific sequence of program counters did not occur during training but occurred while testing, then it would be classified as anomalous.

C. Results

We could detect the program flow being diverted into the **success** function which normally is not executed. That is shown in figure 6 where 4 of the collected program counters are within the range of the **success** function (orange area of the plot) and none of the collected program counters is within the **no_cookies** function range (sky area of the plot), lying above the success range. Red vertical areas indicate sequences not found in the training data, indicating anomalous behaviour. This indicates the confirmed anomaly points that is lying in the horizontal success and vertical red regions.

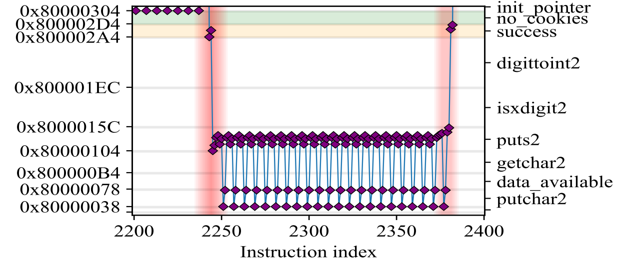

Fig. 6. Timeline of program counters during the anomalous run of stack-mission program.

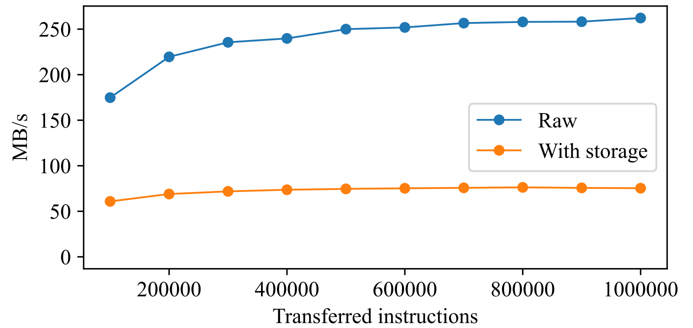

Fig. 7. Throughput depending on the number of transferred items.

Maximum throughput of data transfer we achieved during our tests was 265MB/s ("Raw" in figure 7), we measured the alternative throughput which took into account the time spent copying the whole buffer into separate storage before initiating a new DMA transfer, to avoid overwriting and losing the previous transfer ("With storage" on figure 7), maximum achieved throughput of this type was 75MB/s.

IV. Conclusion

We have proposed a lightweight continuous monitoring system (CMS) based on hardware-software co-design that communicates with the RISC-V to identify any abnormalities in its operational behaviour. An anomaly detection program is being executed by the ARM (Software) that enables continuous evaluation of the system operation to spot hardware failure or unusual system behaviour. The complete design has been prototyped and verified on Zynq FPGA ZC706.

REFERENCES

[1] D. E. Owen, J. Joseph, J. Plusquellic, T. J. Mannos, and B. Dziki, "Node monitoring as a fault detection countermeasure against information leakage within a risc-v microprocessor," *Cryptography*, vol. 6, no. 3, 2022. [Online]. Available: https://www.mdpi.com/2410-387X/6/3/38

[2] D. F. Koranek, S. R. Graham, B. J. Borghetti, and W. C. Henry, "Identification of return-oriented programming attacks using risc-v instruction trace data," *IEEE Access*, vol. 10, pp. 45 347–45 364, 2022.

[3] 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.

[4] S. Lu and R. Lysecky, "Data-driven anomaly detection with timing features for embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 3, apr 2019. [Online]. Available: https://doi.org/10.1145/3279949

[5] 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.

[6] R. N. M. Watson. CHERI-RISC-V, year = 2022, url = https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-risc-v.html , urldate = 2020.

[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] 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.

[9] Z. Lan, L. Xu, and W. Fang, "Fdnn: Feature-based deep neural network model for anomaly detection of kpis." IEEE, 10 2019, pp. 286–289.

[10] 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.

[11] ——, "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.

[12] C. Malone, M. Zahran, and R. Karri, *Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs?* ACM, 2011.

[13] R. Watson, B. Davis, W. Filardo, J. Clarke, and J. Baldwin. (2022) Cheri-exercises. [Online]. Available: https://ctsrd-cheri.github.io/cheri-exercises/missions/uninitialized-stack-frame-control-flow/index.html

[14] 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.

[15] 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.