
AFRL-RI-RS-TR-2021-154

# DECOUPLED REAL-TIME PROGRAM EXECUTION STATE MONITORING

IOWA STATE UNIVERSITY OF SCIENCE AND TECHNOLOGY

*SEPTEMBER 2021*

FINAL TECHNICAL REPORT

***APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED***

STINFO COPY

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

# NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2021-154 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

**/ S /**
SERGEY PANASYUK
Work Unit Manager

**/ S /**
JAMES S. PERRETTA
Deputy Chief, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

# REPORT DOCUMENTATION PAGE

*Form Approved*
**OMB No. 0704-0188**



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| SEPTEMBER 2021 | FINAL TECHNICAL REPORT | APR 2020 – APR 2021 |

**4. TITLE AND SUBTITLE**

DECOUPLED REAL-TIME PROGRAM EXECUTION STATE MONITORING

**5a. CONTRACT NUMBER**
N/A

**5b. GRANT NUMBER**
FA8750-20-1-0504

**5c. PROGRAM ELEMENT NUMBER**
AF Other

**6. AUTHOR(S)**

Akhilesh Tyagi
Henry Duwe

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**
R2ZJ

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Iowa State University of Science & Technology
515 Morrill Rd, 1350 Beardshear Hall
Ames IA 50011

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Air Force Research Laboratory/RIGB
525 Brooks Road
Rome NY 13441-4505

DARPA
675 N. Randolph St
Arlington VA 22203-2114

**10. SPONSOR/MONITOR'S ACRONYM(S)**
AFRL/RI

**11. SPONSOR/MONITOR'S REPORT NUMBER**
AFRL-RI-RS-TR-2021-154

**12. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
The focus of this project is to localize the program execution state from off-processor-chip side-channel sensor streams that are naturally created by the program execution (last level cache -LLC- misses). The side-channel sensor streams evaluated in this project are (1) LLC miss address stream, (2) processor domain power stream, (3) DDR memory domain power stream captured through electromagnetic (EM) emission, and (4) performance monitoring unit (PMU) stream. The localization is performed at a blob level to manage the sampling and computational demands on the power side-channel. An anomaly is detected in the execution when the two consecutive detected paths cannot occur in the golden model of the program. The monitor is evaluated on a Xilinx Zynq Ultrascale+ XCU 106 board which contains two ARM processors and a sea of FLGA fabric. The monitor uses these trained ML models to classify the sensor stream data into a Blob/Path. The multiple streams' classifications are resolved into a single Blob/Path localization based on confidence values of each stream classification. Individual stream's classification accuracy ranges from 80-90%.

**15. SUBJECT TERMS**

Off-processor-chip, side-channel sensor streams, blob level, program execution state integrity (PESI), multi-modal side-channel streams

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | **SERGEY PANASYUK** |
| U | U | U | UU | 35 | 19b. TELEPHONE NUMBER (*Include area code*) **N/A** |

# Table of Contents


**1 Summary** **1**

**2 Introduction** **1**

**3 Methods, Assumptions, and Procedures** **3**
3.1 Evaluation Platform 3
3.2 Blob Creation 4
3.2.1 Strongly-connected component approach 5
3.2.2 SCC approach with dynamic profiling 5
3.2.3 Naive direct DFS profiling approach 6
3.2.4 Natural loops approach 7
3.3 Power Stream 9
3.3.1 Power Domains 9
3.4 LLC Miss Address Stream 11
3.4.1 Explicit State Triggers 13
3.5 Performance Counter Stream 14
3.6 Assumptions for Current Evaluation 14
3.7 Monitor 15

**4 Results and Discussion** **15**
4.1 Benchmark 15
4.2 Blob Characteristics 16
4.3 Power Stream Blob Path Identification 20
4.4 Address Stream Blob Path Identification 23
4.5 PMU Stream Blob Path Identification 23
4.6 Monitor Blob Identification and Anomaly Detection 24
4.7 Discussion 27

**5 Conclusions** **28**

**References** **29**

**LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS** **30**

# List of Figures


1 Monitor Overview....3
2 Xilinx Zynq Ultrascale+ ZCU 106 Platform ....4
3 Blob creation process....5
4 Blob graph naively constructed from profiling. ....7
5 Two Power Domains Affected by Cortex A53 Instruction Execution ....9
6 Shunt Resistance Bypass Board....10
7 Processor Power Domain and DDR Power Domain Acquisition Setup ....11
8 Processor Power Domain and DDR Power Domain Example Power Samples ....11
9 Coherent Cache Interface for Address Stream Collection ....12
10 Example Stream benchmark address trace....13
11 Raw Data Collection Phase for LLC Miss, Power, PMU Data....15
12 Natural Loops Approach: Control Flow Graph With Blobs and Cache Misses ....17
13 Natural loops approach: blob level graph ....18
14 Natural loops approach: static instructions per blob....19
15 SCC approach with dynamic profiling: blob level graph....19
16 Machine Learning Model Approach ....21
17 Path-based Identification data structure ....22
18 Token Runs for each class variable....22
19 Blob-based Identification data structure ....22
20 Confusion Matrix For Address Decision Tree Classifier Model....24
21 State transition at Blob boundary....25
22 Results for Blob-level localization (anomaly-free data) ....26
23 Results for Blob-level localization (anomalous data) ....26
24 Results for Path-level localization (anomaly-free data)....27


# List of Tables


1 Natural loops approach: average dynamic statistics per blob ....18
2 Results on Blob-based Identification ....23

# 1 Summary

The focus of this project is to localize the program execution state from off-processor-chip side-channel sensor streams that are naturally created by the program execution (last level cache - LLC- misses). The side-channel sensor streams evaluated in this project are (1) LLC miss address stream, (2) processor domain power stream, (3) DDR memory domain power stream captured through electromagnetic (EM) emission, and (4) performance monitoring unit (PMU) stream. The localization is performed at a blob level to manage the sampling and computational demands on the power side-channel. This also manages the monitoring overhead trade-off with the localization granularity.

A blob is a program level entity whose boundaries are detectable off-processor through side-channel streams. Blob size can be parametrized in the blob creation heuristic. Typical blob sizes we have encountered are 200 instructions as static size and 100s of million dynamically executed instructions. The goal of the decoupled monitor is to flag a specific path within a blob. An anomaly is detected in the execution when the two consecutive detected paths cannot occur in the golden model of the program.

The monitor is evaluated on a Xilinx Zynq Ultrascale+ XCU 106 board which contains two ARM processors and a sea of FPGA fabric. We target the Cortex A53 processor for the program state localization.

Each of the LLC address, execution path power, performance monitoring unit streams builds machine learning (ML) models for all the paths in a program. The monitor uses these trained ML models to classify the sensor stream data into a Blob/Path. The multiple streams' classifications are resolved into a single Blob/Path localization based on confidence values of each stream classification.

Individual stream's classification accuracy ranges from 80-90% for the Blob/Path classification. The overall execution state localization is evaluated on a benchmark program "stream" with 3 normal execution runs and 2 anomalous runs. The accuracy of this localization is 87.5% for normal runs and 100% for anomalous runs.

# 2 Introduction

This project detects program execution state integrity (PESI) violations. The control flow integrity (CFI) constitutes a subset of PESI violations. Traditional adversary models for program integrity monitoring limit themselves to software channel induced attacks such as code injection/buffer overflow. This work also considers hardware adversaries. These can include a memory/peripheral bus tampering adversary. An extraneous, malicious behavior embedded into a printed circuit board (PCB) to tamper the bus traffic is also within the scope.

Another novelty of the proposed effort is that monitor is decoupled, real-time, and is realized in hardware. The monitor captures the execution program state through side-channels - specifically power, last level cache (LLC) miss address, and performance counter side-channels. Traditionally, a monitor is coupled to the monitored program through computing channels designed to share computing resources. Our remote monitor leads to stronger isolation of the monitor - malware designed to travel through power side-channel or microarchitecture side-channel does not exist at this time. These lightweight taps into the program state also pose minimal overhead on the program

execution environment. This is especially well-suited for real-time programs.

Moreover, the monitor itself is designed to function in real-time. This leads to our decision to implement the monitor in an FPGA environment. Design choices are driven by the real-time response goal.

The power side-channel capabilities are challenged with a modern COTS processor executing at 4GHz clock rate with 4-6 wide speculative, out-of-order, superscalar microarchitecture with 100-200 instructions in flight. Reconstruction of the instruction stream at instruction level granularity through power side channel is not feasible with state of the art. We develop a hierarchical abstraction of the program control-flow graph (CFG) into a blob, consisting of 2000-10000 instructions, and a blob-level control flow graph (BFG) in order to make this problem tractable. The blob boundary defining events need to be visible externally, for example at the memory bus. We chose LLC misses as the primary indicators or triggers for blob boundaries. This reduces the power side-channel problem from instruction stream identification to blob stream identification, resulting in a 2000-10000 XYZ factor reduction in the problem complexity.

Performance counters in a performance monitoring unit (PMU) are typically used for program performance enhancement by identifying the sources of program inefficiency such as cache misses. They have been used to flag program execution granularity at a coarse granularity. Control flow integrity may also be monitored through PMU event counters. In this project, we use PMU counters to establish a blob path identity.

Figure 1 shows the concept overview. The monitor has access to the pre-constructed golden model of the monitored program in the form of its blob level transition graph. The goal of the monitor is to identify Blob/Path identity of the current execution region. Each blob has multiple entry and exit points giving rise to multiple paths. Intuitively, each state in the monitor corresponds to a Blob/Path identity. The incoming multi-modal side-channel sensor streams are analyzed through machine learning classification for a particular Blob/Path identity with a confidence value. The monitor resolves conflicting Blob/Path classifications from different streams through a heuristic that rewards higher confidence values. The PMU stream and boundary power stream are checked only at blob boundaries. They are strong indicators of blob level transition. The LLC miss address stream and window based power stream check the current blob identity periodically based on a sampling window size that keeps sliding forward. The window based periodic sensor streams are also smoothened through hysteresis so that a Blob identity transition is indicated only if it has consistently occurred a few times. The periodic streams are the intra-path identity verifiers. These are good at detecting attacks that hijack control flow for a brief time period and then bring it back to the original execution path. The blob transition streams (PMU and boundary power signatures) are good at detecting anomalies arising from a control flow hijacking that takes the program across blobs.

This effort is a preliminary assessment of the feasibility of multi-modal side-channel streams localizing the program execution at blob path level. This also estimates the practical accuracy and overhead metrics for such a monitor.

The individual side-channel sensor streams show over 80-90% accuracy in path localization. The overall monitor accuracy at normal execution flow localization is 87.5% and at anomalous execution flow detection is 100%.

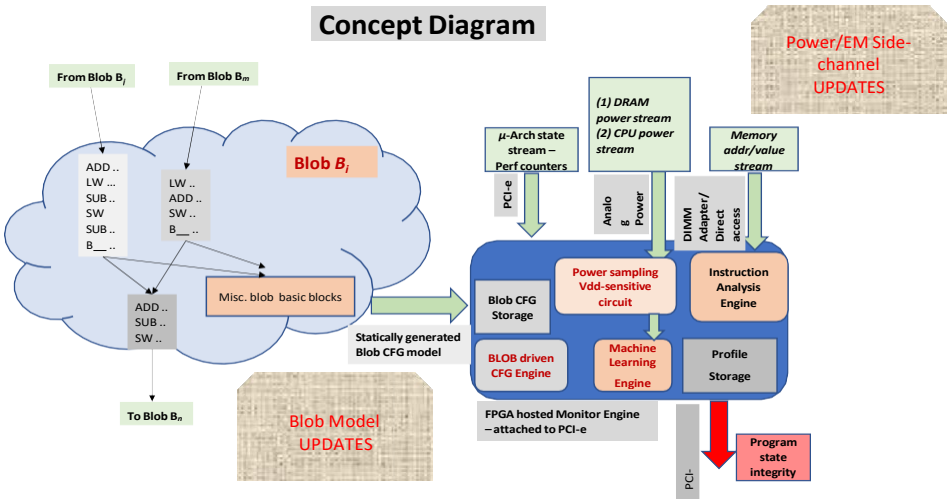



**Figure 1: Monitor Overview**

# 3 Methods, Assumptions, and Procedures

The platform selected for the preliminary evaluation, the program decomposition into blobs, and engineering needed for side-channel taps constitute the project methods.

## 3.1 Evaluation Platform

The following requirements drove our selection of evaluation platform - (1) Easy capture of memory stream during this project phase with the emphasis on the conceptual evaluation of effectiveness of memory stream as an orthogonal resolution channel, and an evaluation of the effect of period of memory bus sampling on the effectiveness of the memory stream in anomaly resolution. (2) Availability of variable complexity processors ranging from 500 MHz - 2 GHz clock, and from 1 core to quad-core; (3) Availability of accelerators to implement monitor logic consisting of Blob transition engine, machine learning engines for various tasks such as power side channel and performance counters, and several other helper tasks; (4) Easily available headers or tap points for needed power domain signals.

We selected Xilinx Zynq Ultrascale+ MPSoC system based board for the evaluation platform, specifically Xilinx EK-U1-ZCU106 category board [3]. The system contains a hard dual-core ARM Cortex-R5, a hard quad-core ARM Cortex-A53, and a programmable logic fabric. Figure 2 shows this board and its floorplan. The targeted applications execute on the quad-core Cortex-A53 [1]. The Cortex-A53 applications are executed on a single core with the other cores shut down in this phase of the work. For future research, the Cortex A53 processor can also host multithreaded applications as a quad-core superscalar microarchitecture.

This setup enables a decoupled monitor for a processor within a SoC. It also offers a platform that emulates a board-level environment in as much as all the monitor state streams except memory are harvested outside processor boundary through interfaces readily available when SoC components are dispersed around the board.

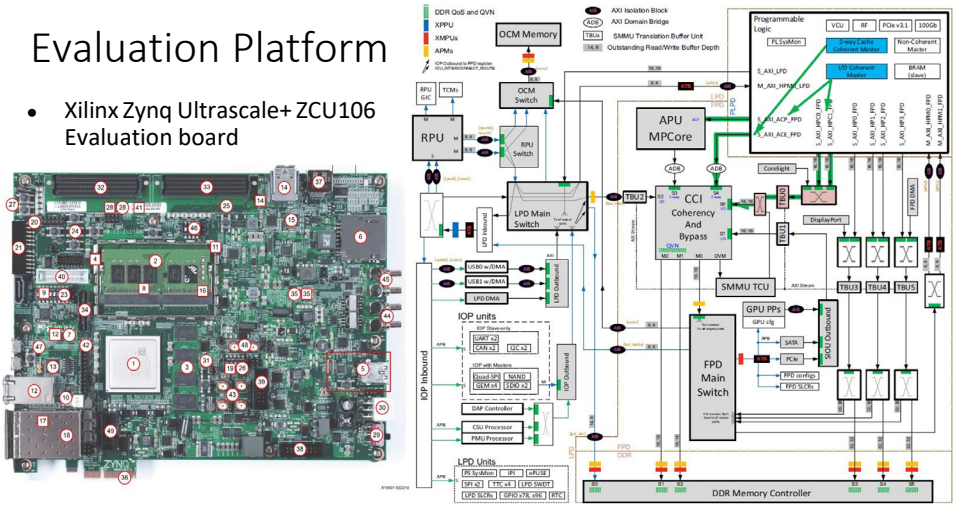



**Figure 2: Xilinx Zynq Ultrascale+ ZCU 106 Platform**

## 3.2 Blob Creation

We explore approaches that can be applied solely through static analysis of the program and also ones augmented with profiling of application execution. Section 3.2 shows the overall approach of blob creation. An application is compiled using Clang to LLVM intermediate representation. LLVM is then used to emit an ARM v8 64-bit assembly file. Assembly files are generated for both a bare-metal target for execution on the evaluation platform as well as Linux target assembly files to execute on gem5. All local labels, etc. are kept as possible entry points of basic blocks in the control flow graph. [1] All machine dependent libraries (e.g.,startup code, wall time libraries, and console output libraries) are wrapped as black-box functions and omitted from static and dynamic analysis. Generated assembly code is then formed into a control flow graph with nodes representing basic blocks and edges representing all possible paths from a basic block, including back edges seen in loops. At this stage of the analysis, relative symbol offsets are used to map all instructions in a basic block to their eventual virtual addresses given the initial load address.

Once a complete linked assembly program is created with sufficient symbol annotation to retain the instruction address to CFG mapping, the final executable binary can be constructed. A blob graph representing the coarsened CFG can be created either directly from the CFG using static analysis or from profiling the executable and collection of runtime information. Profiling can be performed either on Gem5 or on the evaluation platform. One advantage of using gem5 is that future internally available metrics can be used to construct blobs that would otherwise be hard to directly measure in the evaluation platform. Additionally, gem5 provides the ability to run many parallel executions to generate training data when limited evaluation platforms are available. For these reasons we use a gem5 to construct our blob graphs. Benchmark binaries are executed on gem5 running in syscall emulation mode. The gem5 simulator is configured to mirror the evaluation platforms processor. There are 2 layers of caches, with the first level consisting of a 32 kB 4-way set associative data cache and a 32 kB 2-way set associative instruction caches. The

[1] Note that this methodology fits in well with the plans of LLVM [2] to provide cfg information along with assembly for the purpose of link-time assembly code optimizations.

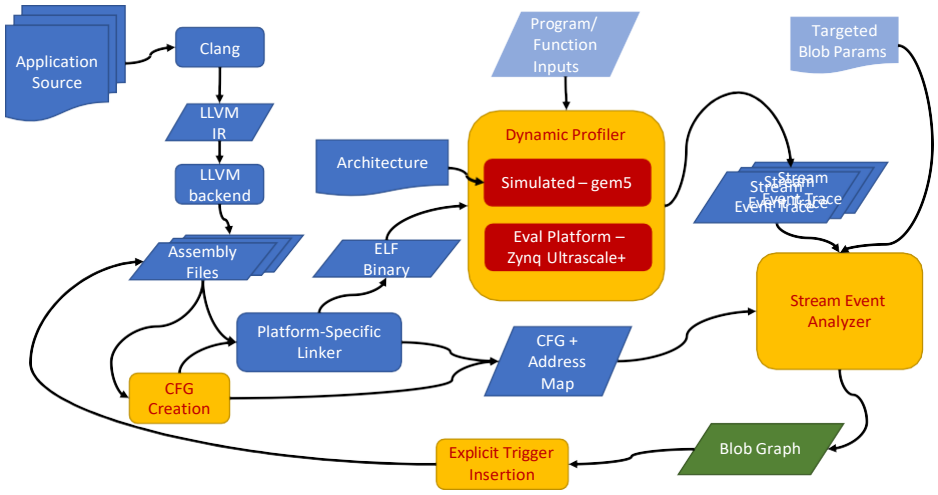



**Figure 3: Blob creation process**

level two cache (last level cache) consists of a 1 MB 16-way set associative cache with a random replacement policy. The simulator reports all last level cache misses (LLC misses), reporting both the address request causing the miss as well current value of the program counter at the time of the miss. After complete execution of the program, the total number of cache misses for each basic block is calculated, and each quantity is inserted into the nodes of the control flow graph data structure. Furthermore, a PC trace consisting of an ordered list of all executed instruction addresses is generated for later use in evaluating blobs.

### 3.2.1 Strongly-connected component approach

Our first approach coarsens the granularity of the control flow by grouping strongly-connected components (SCC) of the control flow graph into blobs. SCCs describe a set in a directed control flow graph in which every node is reachable from every other node. By constructing SCCs out of basic blocks (BBs), the normal control flow of a program can be split up into logical sections—loops, setup, and other function calls. We used a standard depth-first search (DFS) implementation of Kosaraju's algorithm for generating a set of all SCCs in the control-flow graph. Each resulting SCC is labeled as a blob.

While this approach can be done statically in linear time, the resulting blobs have two main drawbacks. First, some blobs would be overly large. For example, if multiple functions are called in a loop, those functions and the loop are an SCC—since there is a way for each basic block to reach the previous one. Second, isolated basic blocks end up in their own SCC with limited or no external signals. The execution of these blobs is likely to be invisible to a decoupled monitor.

### 3.2.2 SCC approach with dynamic profiling

To combat issues with overly large blobs and invisible blobs, profiled cache miss data was used. Specifically, we limit the size of blobs using a threshold of cache misses. If during creation of a blob this threshold is met, a new blob is created using the remainder of the SCC. Additionally,

if blobs are created with no cache misses, meaning they are undetectable, they are merged into other existing detectable blobs. While this method is effective at creating appropriately-sized and externally visible blobs, it often creates blobs that split deeply connected BBs into different blobs resulting in rapid transitions between blobs. If this method was to be pursued further, there would need to be more investigation in splitting blobs based off of function calls or other distinguishable markers.

### 3.2.3 Naive direct DFS profiling approach

The initial approach used to evaluate blobs was to conduct a depth first traversal of the control flow graph, creating new blobs every time a basic block with a threshold number of misses is met. For our tests, we chose a value of 50 as the threshold.

The algorithm (see Algorithm 1) was designed to maximize the natural "depth first" flow of software as each basic block executes. As program execution along a control flow graph generally only moves downwards unless a loop introduces a back edge, traversing in a depth first manner ensures maximum spatial locality. Each consecutive basic block is added into the blob until a memory access monitor would be capable of detecting that the program has changed to a new blob.

Detecting a transition is assumed to require at least 1 cache miss in a new blob, and likely more as a unique pattern in misses will need to be established. A threshold of 50 was chosen after observing that basic blocks with less than 50 cache misses were typically filled with random instruction, cache, and temporary variable mixes. Basic blocks with greater than 50 cache misses seemed to exclusively signal high amounts of temporal access.

**Algorithm 1** Naive Blob Generation

```
1:  function CREATEBLOBS(controlFlowGraph cfg, int threshold_misses)
2:      list blobs ← [ ]
3:      set current_blob ← ∅
4:      node node
5:      while cfg.nodes / blobs do
6:          node ← cfg.next()                          > Iterate graph via DFS
7:          if node.llc_misses >= threshold_misses then
8:              blobs.append(current_blob)
9:              current_blob ← ∅
10:         end if
11:         current_blob.add(node)
12:     end while
13:     return blobs
14: end function
```

With a threshold size of 50, 10 blobs were created. Figure 4 depicts a blob-level control flow graph. Most of the blobs connected back to Blob_0. We determined there were two reasons for this: function calls were being treated as a shared node, with an edge inserted from the caller to the function, and another edge from the function back to the caller. This resulted in many functions having many different edges to different call sites.

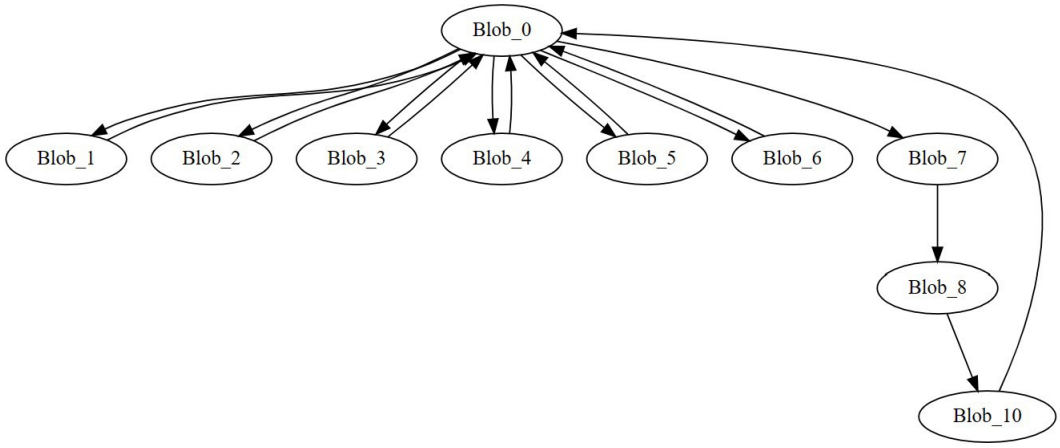



**Figure 4: Blob graph naively constructed from profiling.**

### 3.2.4 Natural loops approach

The naive approach was flawed because there were too many blob transitions, many of which lacked cache misses necessary to classify a transition. We determined that one of the primary reasons for this was that the naive approach did not account for the looping behavior present in many programs. Another issue was that each function call was handled by common nodes in the control flow graph, which introduced a significant amount of density in the graph. A dense graph made for far more edges between blobs, leading to too many paths to efficiently enumerate, as well as too many missed blob transitions.

To address the problems with the naive implementation, we created a new algorithm that considered natural loops present in the code. A natural loop is defined as the smallest set of nodes containing the head and tail of back edge, as well as all predecessors of the head of the back edge though none of the predecessors o the tail of the back edge. A back edge is defined as an edge $n_1$ $n_2$ S.T. $n_2$ **dominates** $n_1$. A node $n_1$ is said to dominated node $n_2$ if every path from the entry node to $n_2$ must first pass through $n_1$.

The blob creation algorithm also optimizes function call handling. Each function call is considered a unique instance, and the nodes of the function control flow graph are inserted recursively at the call site on the control flow graph with a unique ID associated with the function call. This "symbolic inlining" allows function calls to be separated from one another, leading to a far more sparse control flow graph. Uniquely separating function calls also more accurately depicts the actual control flow of the graph, as the preceding conditions prior to a function call differ for each call site, and therefore each call site should be thought of as independent.

After symbolically inlining all function calls, natural loops are then identified. Many natural loops may overlap, such as in the case of nested loops. A combining process is then executed in which overlapping loops are combined to remove overlaps.

**Algorithm 2** Natural Loop Blob Generation

```
1:  function FINDNATURALLOOPS(controlFlowGraph cfg)
2:      loops ← ∅
3:      for Edge e ∈ cfg do
4:          loop ← ∅
5:          if e.end dominates e.start then
6:              loop.add(e.end, e.start)
7:              cfg.reverse()                          > Reverse the direction of all edges
8:              cfg.remove(e.end)
9:              for doNoden ∈ e.end.successors
10:                 loop.add(n)
11:             end for
12:             loops.add(loop)
13:         end if
14:     end for
15:     return loops
16: end function
17: function COMBINELOOPS(controlFlowGraph cfg, set natural loops, int threshold)
18:     visible_loops ← {natural_loops |Node n.misses > threshold }
19:     sort(visible_loops)                            > Sorted by cache misses
20:     for Loop l ∈ visible_loops do
21:         l ← {natural_loops | l.maxNode ∈ loop
22:         natural_loops ← natural_loops  l
23:     end for
24: end function
25: function CREATEBLOBS(controlFlowGraph cfg, int threshold misses)
26:     list blobs ← [ ]
27:     set current blob ← ∅
28:     node node
29:     natural_loops ← FINDNATURALLOOPS(cfg)
30:     COMBINELOOPS(cfg, natural loops, threshold misses)
31:     for Loop l ∈ natural loops do
32:         cfg ← cfg  l                   > Consolidate all natural loops nodes into 1 node
33:     end for
34:     while cfg.nodes / blobs do
35:         node← cfg.next()                           > Iterate graph via DFS
36:         if node.llc_misses >= threshold_misses then
37:             blobs.append(current blob)
38:             current blob ← ∅
39:         end if
40:         current blob.add(node)
41:     end while
42:     return blobs
43: end function
```

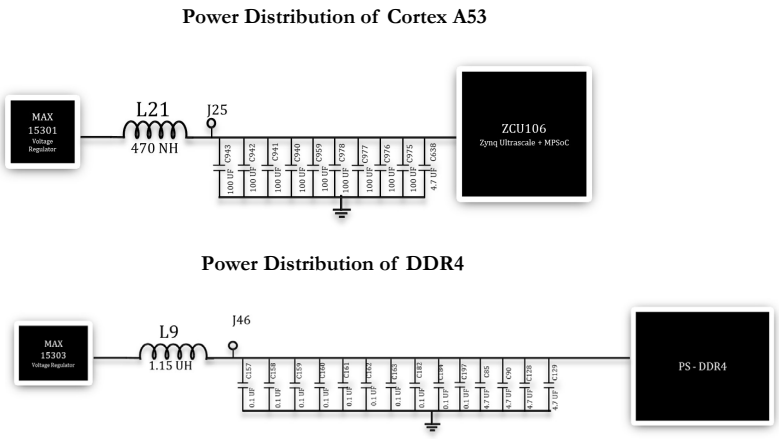



**Figure 5: Two Power Domains Affected by Cortex A53 Instruction Execution**

## 3.3 Power Stream

Circuit level switching activity induced by a specific type of computation and/or data-dependent aspects of computation create a distinct power signature at the $V_{dd}$ pin. Power side-channel taps either a low series resistance to $V_{dd}$ pin or some other point in the power supply network for high frequency sampling. These time domain samples may correlate well with either the secret data or instruction type either in time domain or in frequency domain or in time-frequency domain. Neither of these classical uses of power side channel are applicable in this decoupled monitor scenario. The secret data extraction requires multiple experiments with the same algorithm execution to be able to extract all the bits of a secret key. This is feasible for a cryptographic encryption/decryption program, since it can be invoked multiple times with the same key. Instruction level identification gives only 1 ns with a 1 GHz single core, single issue processor to detect the instruction opcode, which is quite challenging. Moreover, pipelines spread the instruction level signatures over multiple clock cycles over unpredictable instruction schedules and ordering of events.

In this project, we strive to use power signatures of an execution path to infer if the program execution state is going through a specific path within a blob. For a path of hundreds of instructions, despite the pipeline spreading of the signature, this identification is much more feasible. The time available is hundreds of times larger as well. The path level power signatures also have lower variance due to the law of large numbers.

### 3.3.1 Power Domains

The Cortex A53 has two distinct power domains that are affected by the instruction execution. Figure 5 shows these domains. A processor power domain supplies the processor computational cores. A DDR domain supplies the power to the memory. When an instruction executes, it will affect the DDR power domain whenever there is a read or write miss in caches leading to memory activity. This could happen on an I-cache miss during instruction read or during data read/write in a load/store instruction. Every instruction activates various components of the pipeline with variable activation schedules. This activity is reflected in the processor domain. This is the rationale behind capturing a combined processor domain and DDR domain signature of a path through blob.

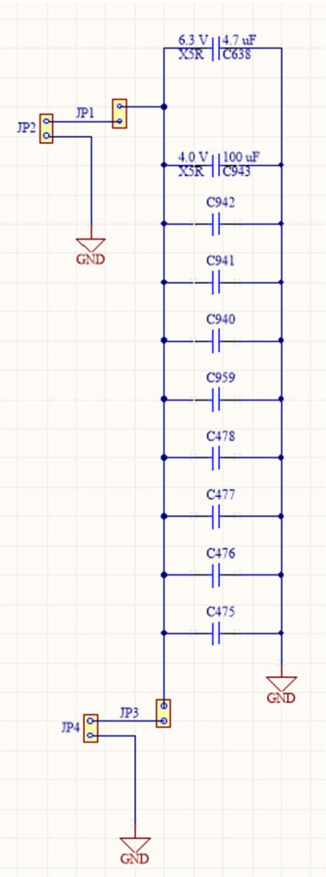



**Figure 6: Shunt Resistance Bypass Board**

The processor power domain has additional circuitry on the board to maintain the supply voltage within a small range so that $V_{dd}$ IR drops do not disable the processor. As shown in Figure 5, the processor power domain has capacitance of the order of 900 $\mu$F along with a power management IC. This large capacitive bank can source sufficient charge to mute the instruction level execution signature peaks and valleys within the *RC* constant of the circuit. There is a jumper that can be used to attach the digitizer probe. However signal variation seen at this point with all the 900 $\mu$F capacitance is not significant enough for path identification.

We removed capacitors from this capacitor bank in 100 $\mu$F steps to see what effect it might have on the voltage management unit. If more than 400 $\mu$F of the capacitance is removed, the voltage supply becomes dysfunctional. We do see more of a instruction and data dependent signal variation with 400 $\mu$F of the removed capacitance. It however still is not as pronounced as it could be with a small shunt resistance across which a probe could be used for a digitizer. We designed and fabricated a small printed circuit board that allows for such a shunt resistance along with all the 10 capacitors of 904 $\mu$F. It is shown in Figure 6.

The DDR power domain is monitored through electromagnetic side-channel since there was no place on the board to tap it. TBPS01 EM Probe available fro Amazon for less than $100 is used to monitor the memory bus activity. The probe from the EM sensor is attached to the same

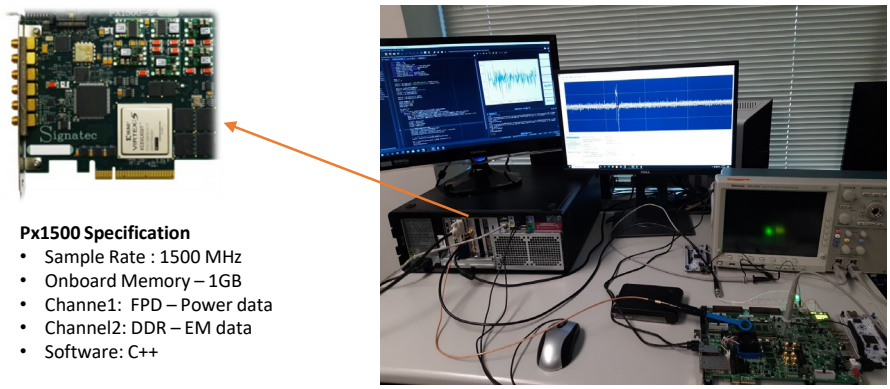



**Figure 7: Processor Power Domain and DDR Power Domain Acquisition Setup**

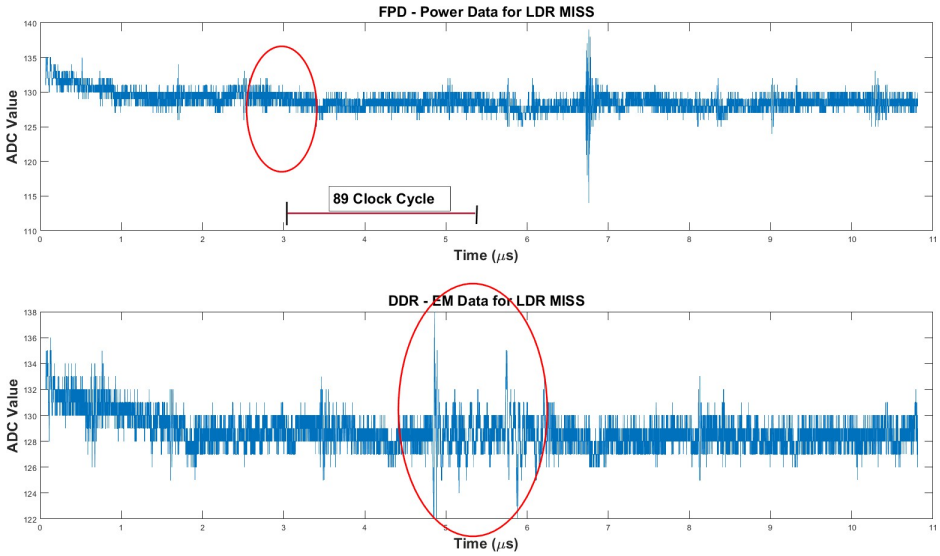



**Figure 8: Processor Power Domain and DDR Power Domain Example Power Samples**

digitizer that samples the processor power domain. We use Signatec PX1500 digitizer. It has a peak sampling rate of 1.5GHz. It is a good practice to oversample each targeted clock cycle by a factor of 10-20 so that important features are not missed. This gives the highest processor frequency that can be effectively monitored in the range 75-150 MHz. It has 1GB onboard memory to buffer 8-bit samples. Channe1 1 is connected to the processor power domain. Channel 2 is connected to the EM probe monitoring the DDR domain. Figure 7 shows this setup. Figure 8 shows an example power sample stream for both processor domain and DDR domain.

## 3.4 LLC Miss Address Stream

Since Zynq ZCU106 is an SoC platform, it allows for new behavior/core to be created in the FPGA fabric of LUTs. This core can also have coherent access to the memory with a shared view with the other cores of the Cortex-A53. Towards this end, Xilinx provides an interface with cache coherence

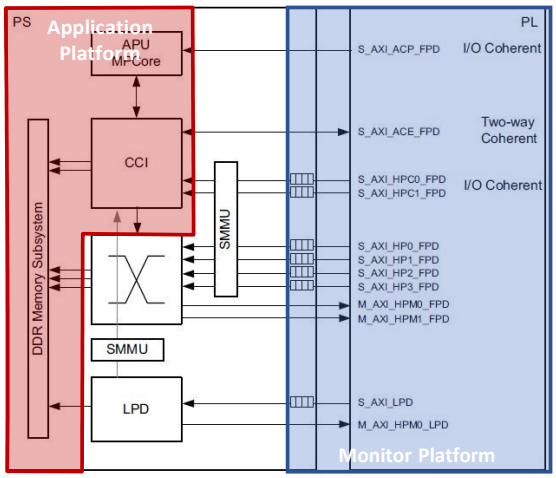



**Figure 9: Coherent Cache Interface for Address Stream Collection**

fabric through cache coherence interface (CCI) for the LUT based memory client. Figure 9 shows this CCI interface. When an LLC miss occurs from one of the cores of Cortex A53, it shows up at the cache coherence common bus connecting all the memory clients. This address (and data) is broadcast to all the clients through CCI. This allows for the LUT based memory client to garner the address (and data) from the CCI interface giving us an easy access to the memory address stream.

Figure 10 shows an example address trace from the Stream benchmark with the overlaid ground truth blobs. We can see that each blob appears to have a unique pattern for the sequence of addresses. This is true across multiple executions of the same blob (e.g., blob 4) even for different entry points to the blob (e.g., blob 30 and 31). However, sometimes a different path through the blob may produce a very different address trace (e.g., blob 32 vs blob 30 and 31).

In order to produce tokens that continually localize the program execution at blob/path level, last level cache misses addresses are recorded through the CCI. Additional type flag indicating the memory operation type (read, write, and maintenance operations) is also kept. The input cache stream is first analyzed for outliers. Outliers are defined by any data point that has a modified Z-score greater than 2. Outlier filtering removes many stray miss values that seemed to decrease the accuracy of classification into a blob and blob path.

After removing outliers, the cache miss data is windowed into 100 sequentially adjacent cache misses. A value of 100 was selected experimentally to optimize the tradeoff between prediction latency and prediction accuracy. Each set of 100 sequential cache misses are then binned based on access type. A series of 50 symmetric bins distributed across the address range of the data for both memory read and write accesses. The range is determined by the minimum and maximum outlier cutoff values, calculated by $2 .\sigma \pm \mu$. Bins are filled by counting the amount of cache misses in each bin range across the 100 cache miss wide window. Finally, all other remaining cache misses in the window are summed in a final bin of the vector.

All feature vectors are generated over non-overlapping windows of the cache channel. The vectors are then used to train a random forest model consisting of 5 decision trees. This model was chosen as it provides a probabilistic prediction with low prediction and training latency.

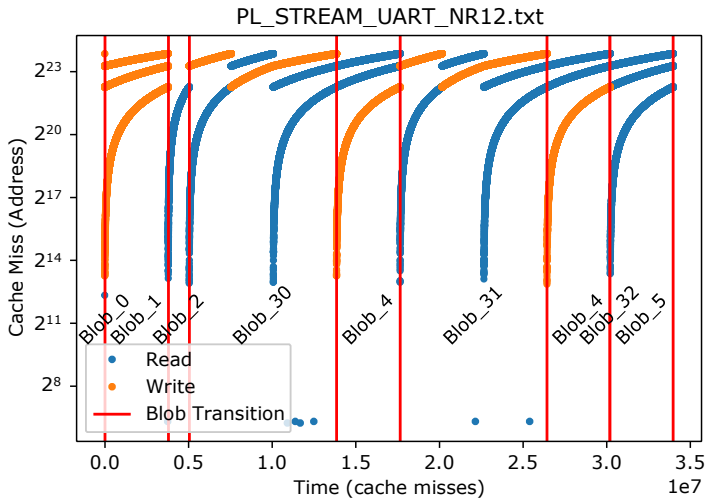



**Figure 10: Example Stream benchmark address trace.**

### 3.4.1 Explicit State Triggers

Although blob creation based on externally-visible signals that an application inherently generates is desirable because it does not require application modification, it lacks the ability to control for disambiguation of paths within or across blobs or latency of anomaly detection. As a fall-back explicit state triggers can be inserted into the application executable. These explicit state triggers directly cause an externally-visible signal to be generated at a particular point within the blob graph. We insert these signals directly into the application assembly by redirecting known BB target symbols. We use two main types of explicit triggers – (1) causing an artificial cache miss of a known, already used address via a cache flush instruction and (2) writing to a known, but reserved address.

In order to leverage a cache flush instruction (e.g., CIVAC in ARM), we must identify a particular address that we reliably expect to be accessed by the application at the time of the trigger. Therefore, we select an instruction to flush since fetching of instructions is strongly dependent on the location within the CFG (and thus blob graph). While data accesses can be correlated to locations within a blob graph they are more likely to be aliased across different points since the same data locations are often accessed in multiple locations within a program. As a prototype, we flush the instruction immediately after a function call (i.e., the return address of a function). We instrument the call to flush the cache block pointed to by the return address. Then, when the matching return instruction is executed, the flushed cache block is fetched resulting in the externally-visible signal. Such an approach is relatively low-overhead per trigger (roughly two added instructions and an additional cache miss) and does not require any reserved memory location.

A more flexible approach selects a reserved address. The reserved address is intentionally flushed from the cache when the application is loaded. At each trigger point in the blob graph the reserved address is written to and then flushed. This is advantageous in that additional information can be externally signaled. This is the approach we use to generate the performance counter stream (see Section 3.5).

## 3.5 Performance Counter Stream

All modern processors support a performance measurement unit (PMU). This unit consists of programmable counters. Some set of microarchitecture level events that affect performance can be counted during a program execution. These events can include L1 instruction cache #misses, #retired instructions, #issued instructions, clock cycles count, and #branch mispredictions. They are a separate hardware unit enabling continuous overhead-free monitoring without posing an overhead on the application execution. A small number of performance counter registers are supported. For instance, ARM Cortex A53 supports 6 physical counters. Each counter register can be bound to a specific event through programmable registers. For instance, counter 0 could be associated with #L1 D-cache misses.

ARM provides low level instructions to manage the PMU interface. For instance, `ARM_PMU_Enable` and `ARM_PMU_Disable` enable and disable the PMU. Similarly, `ARM_PMU_Get_CCNTR` reads a performance counter. PMU counter values are read at each blob boundary and then these values are written to a specific memory address that is predefined by the CCI. When the target application is running on the platform, the CCI will constantly monitor the predefined memory address. All memory writes to that memory address will be captured by the CCI and the CCI will generate a set of markers which includes the corresponding PMU counter values. This set of markers will be put into the address stream and then later consumed by the monitor.

We use all of the six programmable PMU event counters and the non-programmable PMU cycle counter to capture the CPU activities during the execution time. The six programmable PMU events we picked are: #instruction retired, #L1D cache access, #L1I cache access, #predictable branch speculatively executed, #Load retired, and #bus cycles. We construct our PMU counter vector by using these seven counters. By doing so, the characteristics of an execution path in all of the blobs can be abstracted by a PMU counter vector.

The machine learning training data are collected by running the golden modal program multiple times. In the training data, each PMU vector is labeled by its corresponding path ID in a blob. Thus, different paths within the same blob are labeled differently. Due to the time constraint imposed by a real-time system, we decided to use SVM linear classifier, a fast machine learning model which does not bring too much overhead to the system, yet remains effective in classifying PMU counter vectors with a high accuracy. The machine learning model is integrated into the monitor. Upon seeing a PMU vector in the address stream, the machine learning model will classify this vector into a path which exists in the golden model. This path level classification is accompanied by a confidence value.

## 3.6 Assumptions for Current Evaluation

Once the multi-modal stream data is collected, it needs to be analyzed. For instance, the power samples after appropriate preprocessing such as principal component analysis (PCA) would be classified into one of the blob paths. Performing this computation in the time needed for a path execution, likely of the order of a few hundred instructions, is challenging. Since this phase is a feasibility study, the data collection activity was decoupled from the data analysis activity. Figure 11 shows the process of collecting the raw data from an execution of a program for power, LLC miss address, and PMU counter streams. The power raw data is collected through the PX1500 digitizer. This data is then curated and classified for a Blob/Path pair with a SVM linear multi-class

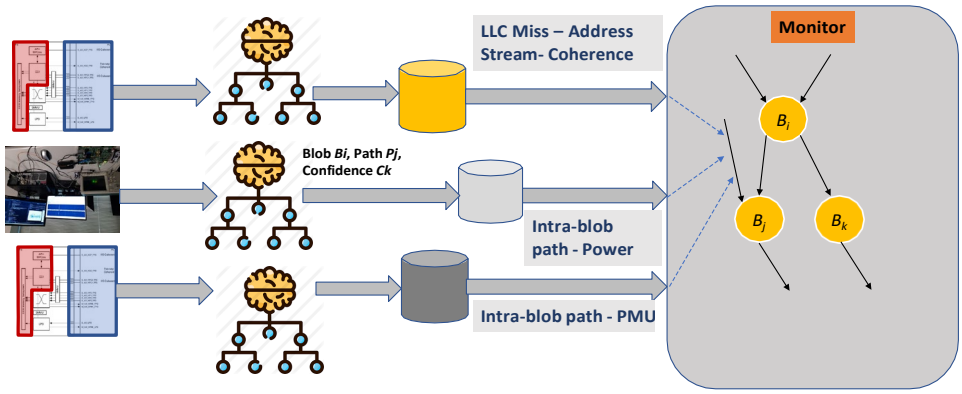



**Figure 11: Raw Data Collection Phase for LLC Miss, Power, PMU Data**

model. The power stream with the Blob/Path pair, confidence value is then archived on a disk. Same holds for the LLC miss address collected through the cache coherence fabric on Xilinx Zynq ZCU 106. The PMU data collected through a PMU counter register read instruction is written to a special address which is then captured by the cache coherence fabric. The monitoring phase walks through these synchronized streams of Blob/Path and confidence value tokens.

The monitor walks through these tokens in a synchronized manner and makes the decision on the current localization in the program execution (specific blob/path pair). In case of an anomalous execution, it flags an anomaly in the program execution flow. Since the preprocessed streams contain post-ML tokens, the computation needs for the monitor are just for the decision and backtracking heuristics. This is sufficient to establish the feasibility of the scheme.

### 3.7 Monitor

Monitor is the heart of this system to aggregate the incoming data coming through multi-modal sensor streams into a decision about blob and path level localization. When this localization is inconsistent with the golden model of the blob level graph, an anomaly is flagged.

This monitor is built as a state machine where each state corresponds to a blob/path. The raw data from the sensor streams is reduced to tokens representing blobs and paths. This is done through trained machine learning models. The monitor consumes these blob/path token sequences that come through the PMU, address and power streams. The key activity it must undertake is to synchronize these tokens from the three streams. A notion of time is maintained based on the programmable logic generated timestamps, as explained in the next section.

## 4 Results and Discussion

### 4.1 Benchmark

The first target benchmark selected was the STREAM memory bandwidth benchmark. This benchmark is an industry standard test of sustained memory bandwidth in high performance computing environments. The benchmark was selected because of the high amount of last-level cache misses and memory accesses that are generated by the benchmark.

The benchmark consists of a set of vector operations executing on arrays that are at least 4x the size of the systems last level cache. Our experimental setup was configured with a vector size of 10 million 64-bit elements. The benchmark functions by executing a series of vector operations on the large arrays present in the system. These operations include an array copy, scalar multiplication, vector addition, and a triad operation. The benchmark allows a configurable number of iterations to be executed, set to 2 to allow full coverage of the code while minimizing runtime.

The benchmark is run in single core mode, with OpenMP directives disabled. Modifications have been made to the timing system used to produce benchmark results, removing the dependency on Linux wall-clock libraries and instead using Xilinx system timers.

Executing STREAM on gem5, a total of 33,632,910 data memory last-level cache misses were recorded per run. Execution took 14,250,665,515 clock cycles to complete, leading to a total simulated time of 1.42 seconds.

## 4.2 Blob Characteristics

The natural loop blob creation algorithm (see Section 3.2.4) was executed on the STREAM benchmark and generated 5 blobs. Blobs had an average of 211.3 static instructions per each blob. Figure 13 shows blobs overlayed over the control flow graph. Functions are inlined in this representation, meaning each function call is represented by a unique set of nodes on the graph. Nodes with greater than 50 cache misses (the threshold value used in the algorithm) are highlighted in red.

Blobs were then dynamically analyzed using a program counter trace generated by gem5 during the dynamic analysis phase. The trace contains an ordered list of all instruction addresses in the order they were executed, along side time stamps that allow synchronizing the program counter trace and the DRAM access trace. Dynamic analysis was able to confirm that in a non-anomalous run, program execution conformed to our blob-level graph (Fig. 13).

Dynamic analysis was used to count the number of instructions that are executed in each blob access, as well as the total number of blob accesses and the order in which blob accesses occurred. Furthermore, a count of how many instructions are executed after entering a blob but before encountering a cache miss is tracked as an indicator of minimum latency a cache based classifier would encounter (See Table 1). This shows the natural blob creation process has created relatively balanced blobs in terms of dynamic instruction – blobs 1-5 have over 100 million instructions dynamically executed but less than a billion instructions. Blob 0 is the entrance blob which has very few externally visible cache misses and very few executed instructions. Therefore, the bulk of the time is spent in relatively few blobs with few visits each. That is there is not a lot of switching between blobs that may cause noise in a monitor. Furthermore, the transitions between these blobs are within 410 thousand instructions or 0.17% of the start of the execution paths. Running at 1GHz or more this would allow the lag of a monitor to be less than a millisecond in detection latency.

The SCC approach with dynamic profiling provides a larger number of blobs, ranging from 12 blobs up to 287 blobs for stream. However, even for the blob graph with the fewest number of blobs (a threshold of greater than two produces 12 blobs), execution often oscillates between several blobs resulting in the blobs being ineffectual for coarsening the granularity of anomaly detection. Figure 15 shows a portion of such a blob graph highlighting a loop containing about 4.5 million LLC misses with each blob accounting for 1.5 million. While this balance would seem beneficial, inspection of the CFG and interleaving of the misses shows that the body of the loop has

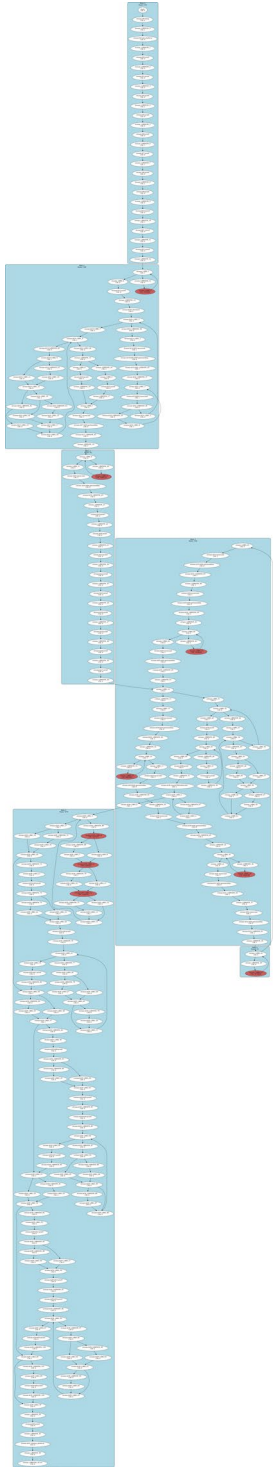


**Figure 12: Natural Loops Approach: Control Flow Graph With Blobs and Cache Misses**

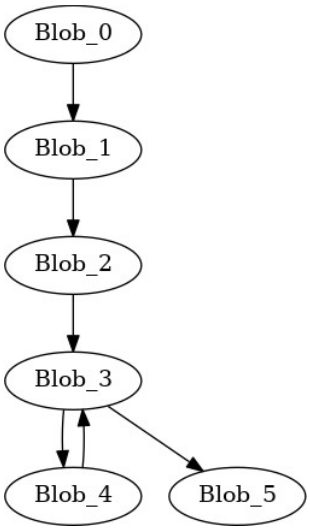



**Figure 13: Natural loops approach: blob level graph**

**Table 1: Natural loops approach: average dynamic statistics per blob**

| | blob | count | Dynamic Instructions | | | Inst. Before Miss | Inst. After Exit |
|---|---|---|---|---|---|---|---|
| | | | mean | min | max | mean | mean |
| 0 | Blob_0 | 1 | 14241 | 14241 | 14241 | 0 | 11 |
| 1 | Blob_1 | 1 | 260400425 | 260400425 | 260400425 | 11 | 343634 |
| 2 | Blob_2 | 1 | 170002890 | 170002890 | 170002890 | 343634 | 6 |
| 3 | Blob_3 | 3 | 400284005 | 31757 | 600410146 | 102497 | 273254 |
| 4 | Blob_4 | 2 | 240153748 | 240000020 | 240307477 | 409879 | 153742 |
| 5 | Blob_5 | 1 | 740003339 | 740003339 | 740003339 | 5 | 0 |

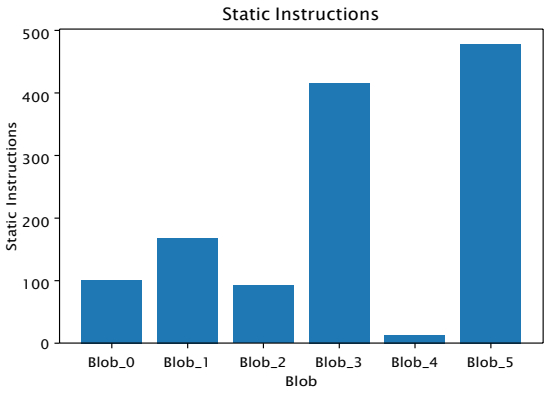



**Figure 14: Natural loops approach: static instructions per blob**

been broken up across different blobs resulting in interleaving of LLC misses at the granularity of a single LLC miss. Therefore, we use the natural loop approach for the remainder of our anomaly monitor evaluations.

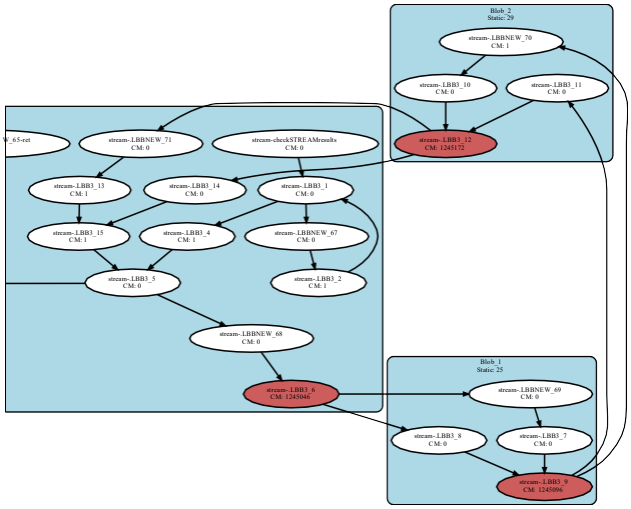
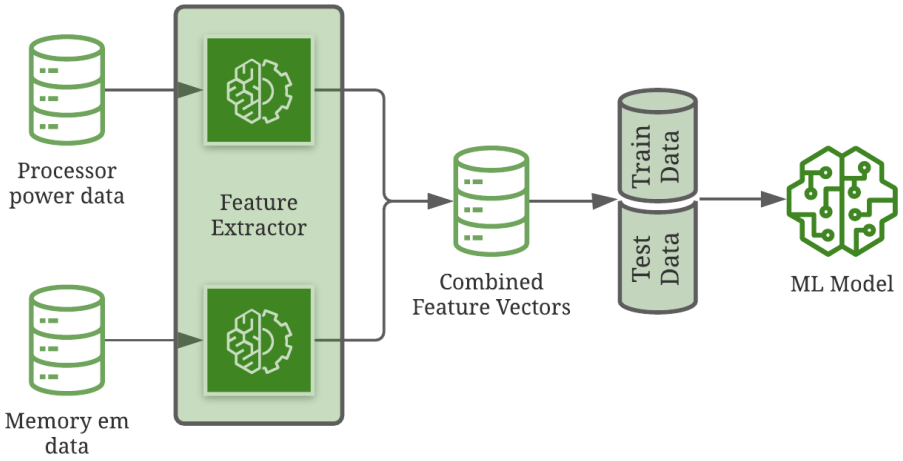


**Figure 15: SCC approach with dynamic profiling: blob level graph**

## 4.3 Power Stream Blob Path Identification

In this experiment, our primary goal is to trace the execution flow of the stream benchmark through power-based sidechannel information and report if it ever differs from the corresponding path in the golden model. We recorded 100 runs of the power side-channel information from the execution of stream benchmark for the golden path to build a machine learning model. The datasets are cleaned to remove the power samples collected during the program suspension interrupt handlers. These power sample sequences are labeled according by the corresponding pathID. The pathID is determined using the PMU marker from benchmark's blob entry for each blob. The PMU marker that emits the PMU counter values also associates the pathID of the blob path just traversed. The process of building a machine learning model is illustrated in Fig. 16.

Feature Extractor is an important step in any classification problem. This project prepares a train/test dataset with Fast Fourier Transform followed by Principal Component Analysis (PCA). The power stream data often fails to synchronize across multiple runs due to irregular delay in interrupts. This causes a significant impact on classifying raw datasets. We convert the data into frequency domain and generate the feature vectors using PCA to avoid this. PCA is a well-known technique for dimensionality reduction. The supervised learning model's size and efficiency such as SVM is severely affected by the feature vector size of the training dataset. PCA identifies and preserves the critical components of the feature vector with 95% variability in the FFT processed data, which helps to reduce the SVM model's training complexity. The feature extractor techniques are applied separately on processor power and memory EM data to preserve their unique features. The feature vectors of processor and memory power data are concatenated for the given path to build the model. We use the Linear-SVM model to identify the blob/path execution localization in the benchmark.




;

**Figure 16: Machine Learning Model Approach**

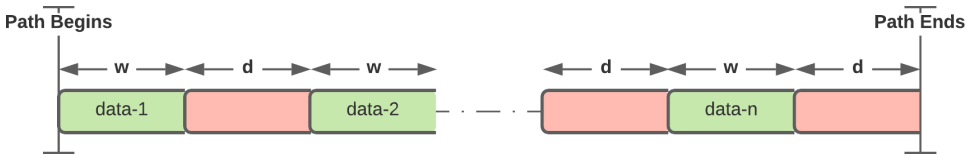
In this work, we have built two different machine learning models on the recorded power stream datasets with the following objectives.

***Path-based Identification:*** The machine learning model is built to localize into a unique path through a blob that the benchmark executes.

***Blob-based Identification:*** The machine learning model is built to localize the execution into a blob in the benchmark.

Though we build two different models, feature extractor steps are identical. The main difference in these models is based on the data used in training. In one case, targeting blob level identity, a power samples window at the beginning of a blob path concatenated with a power window at the end of a blob path constitutes the feature vector. In the other, a fixed size $N$ -samples window slides over the sampled path to determine how likely is the power sample sequence to belong to that path.

**Path-based Identification** The datasets are constructed with base parameters such as Window Length ($w$) and Window Separation($d$). Since the benchmark execution takes several hours, the dataset is converted into multiple windows of width $W$ separated from each other by $d$ samples. This also reduces the host computer memory usage. For the given computational resources, the $w$ and $d$are set to 1000000, which is illustrated in Fig. 17. The constructed data is fed to the feature extractor PCA to compute unique feature values for processor power data and memory EM data. After PCA, the feature vector dimensions are reduced to 4590 from 1000000. The reduced feature vectors are concatenated to train the SVM model for unique pathIDs. The trained SVM model results in 81.7% accuracy on predicting the pathID. The saved model is used to trace the execution of the test run. The Stream benchmark has nine unique paths, including the configuration steps. The PathID "*Blob_0*" and "*Blob_3_3*" have classification accuracy less than 50%, due to shorter execution paths compared to others. The "*Blob_0*" and "*Blob_3_3*" have less memory access than other blobs. The PathID "*Blob_5*" has the highest classification accuracy of 90%, whereas the accuracy of the remaining paths is between 70-80%.




;

**Figure 17: Path-based Identification data structure**

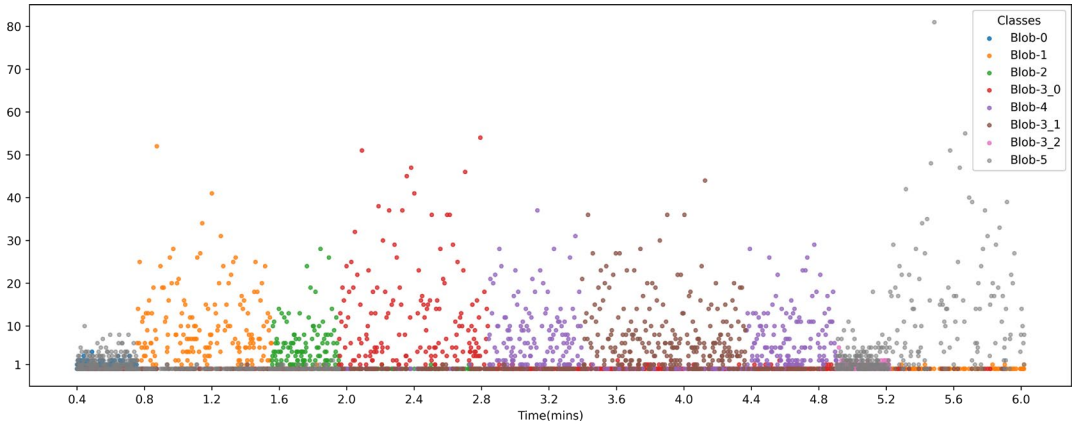
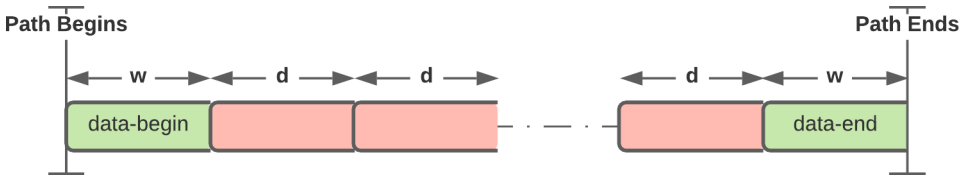



**Figure 18: Token Runs for each class variable**

The SVM-machine learning model is built to predict class variables as a token on the complete execution path of the benchmark along with the confidence values. There are 14056 tokens generated from the results of the ML model. We also computed the runs of each class tokens along the execution path and plotted them in Fig. 18. The x-axis defines the relative time calculated from the timestamp of the PX1500 interrupt marker. The y-axis represents the runs of each class token.

**Blob-based Identification** The datasets are constructed with the beginning and ending windows of power samples for each path. Each path datasets are $2w$ in size as illustrated in Fig. 21.




;

**Figure 19: Blob-based Identification data structure**

We built a power model using the new features obtained from the feature extractor in Fig. 16. The model results in lower accuracy of around 55% to distinguish the blob. The lower accuracy

may be caused by the noisy power samples from the PL interrupts. Later, we recollected the power samples without PL interrupt and rebuilt a similar ML model. The new ML model results in 89.5% success rate on classifying the blobs.

**Table 2: Results on Blob-based Identification**

| Blob-ID | Success rate |
|---|---|
| Blob-0 | 85% |
| Blob-1 | 96% |
| Blob-2 | 95% |
| Blob-3 | 86% |
| Blob-4 | 85% |
| Blob-5 | 90% |

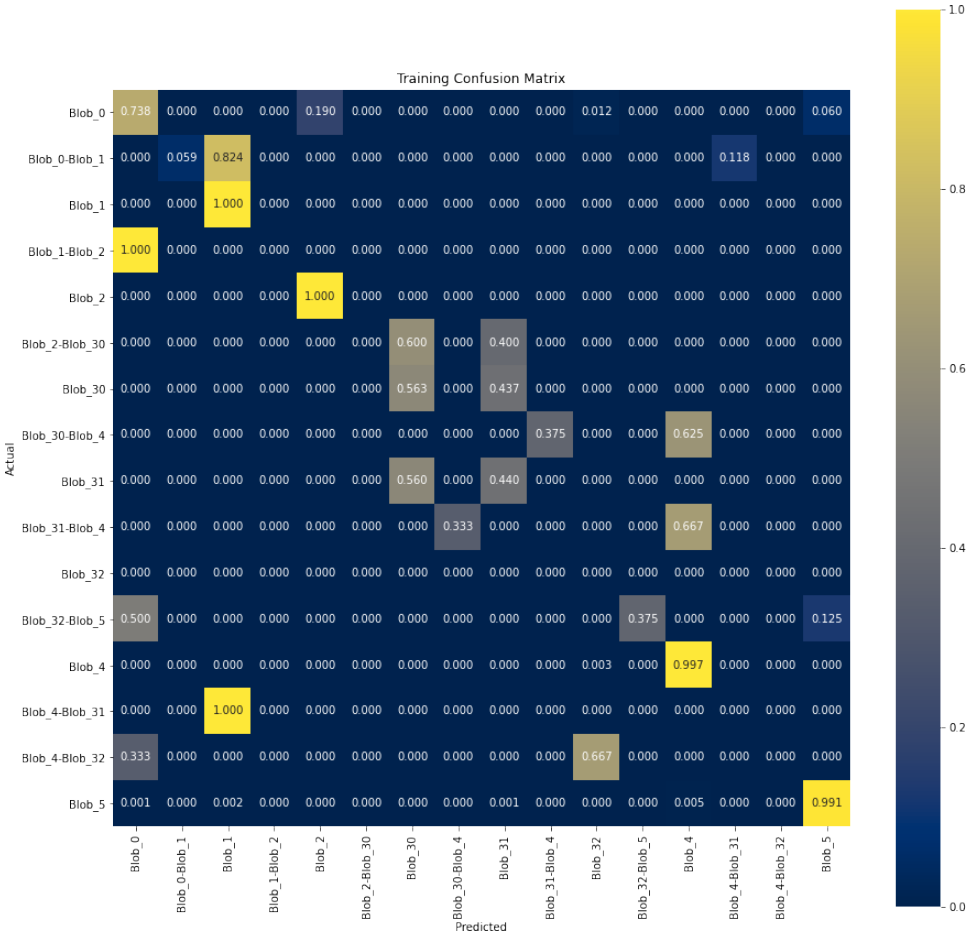
## 4.4 Address Stream Blob Path Identification

A decision tree model was trained using an unlimited depth and unlimited number of leaf nodes. The model was trained using non overlapping windows from a complete non-anomalous run. The model was tested against a second non-anomalous run, selecting 10K random windows as well as 100 random data windows from each class. The model ultimately produced an accuracy of 74.4%.

A confusion matrix (Fig. 20) demonstrates a high level of accuracy for individual blobs. However, attempts to classify transitions lead to less accurate results as did differentiating between different paths. Blob accuracy is lower for smaller blobs and paths as well, notably seen in "Blob_0" and "Blob_32" ('Path_2' through 'Blob_3').

## 4.5 PMU Stream Blob Path Identification

Two different PMU configurations are tested to achieve the highest accuracy. In the first configuration, PMU starts to count the number of events at the same time when the benchmark program starts. At each blob boundary, a PMU vector is generated with its corresponding path ID. We collect the data by running the benchmark 100 times to train an SVM linear machine learning model with half of the collected data (50 runs). The other half of the data is used to test the machine learning model accuracy. The SVM model can classify the testing data with 100% accuracy. During the experiment with different configuration of the PMU, we noticed that the program logic interrupt and PX interrupt used to empty the buffers can affect the PMU counter values. This is because the PMU will continue counting the events during the interrupt. The interrupt handler is just an infinite event loop. It likely affects the total execution cycles event, but not many others. To reduce the noise in our PMU counter values, we tested another configuration where the PMU counter is stopped at entry into the program suspension interrupt handler. At the time, the PMU is stopped, the PMU counters are also reset to 0. The PMU will restart counting when the interrupt ends. This gives an abbreviated signature of the path segment from the last interrupt handler to the next blob. We performed the same machine learning experiment by dividing the data into two halves, one half for training and one half for testing. The machine learning results shows that even by capturing a small path window at the path end, our SVM model can achieve 87% accuracy.



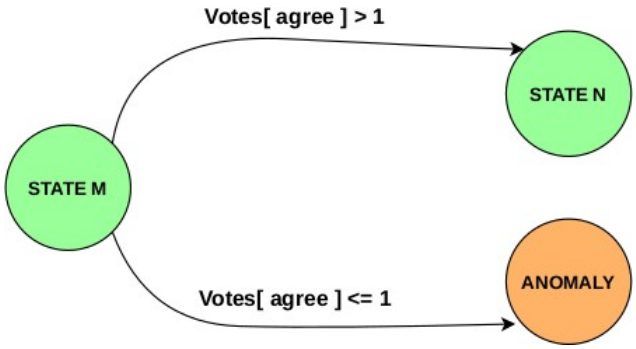

;

**Figure 20: Confusion Matrix For Address Decision Tree Classifier Model**

## 4.6 Monitor Blob Identification and Anomaly Detection

The Monitor is designed to check program state integrity at every blob transition boundary and along the paths within a blob at certain intervals. It does so by maintaining the known control flow paths of the program. This golden path model is checked against the current state of the program for anomalous behaviors. The current state of the program is determined by the unified decision from Power, PMU and Address streams in the form of tokens generated by corresponding machine learning models. A token is a label uniquely identifying a blob/path. In a real time system, the blob/path token generation from the sampled program state will occur as a sublayer within the monitor by invoking the trained ML models.

Since the Monitor consumes tokens from three different streams, it is important to do so in a




;

**Figure 21: State transition at Blob boundary**

synchronized manner. Synchronization is done using a marker in the address stream that uniquely identifies when a PMU counter state is read. Power tokens are also aligned with this unique PMU marker. Since PMU counter state is read at blob boundaries, this technique synchronizes all the streams at every blob entry. Token consumption rate is also an important factor in keeping the three streams synchronized for the Monitor. PMU tokens are the most infrequent and are generated once per blob. Power tokens are generated at a fixed rate which is a function of the sampling frequency of the digitizer. Address token generation is irregular, driven by LLC activity. To allow synchronized token consumption, a counter is implemented within the FPGA logic (PL) that increments at every clock. Every address stream token is timestamped with this counter value. This gives us a sense of time elapsed in a blob in terms of PL clock. Power stream uses this notion of time to timestamp its own tokens, initializing at the address token timestamp at blob entry and incrementing every next token by a fixed amount determined by its sampling rate converted into PL clock domain.

The Monitor maintains its own clock that is initialized on entry to the first blob in the program. A token is consumed when its clock reaches the next closest timestamp value on a Power or Address token. This continues until the next blob entry at which point a PMU token is consumed.

In the current state of the Monitor, anomaly checks are performed at two granularities:

- At blob boundaries
- Along a path

On entering a new blob, the Monitor checks for anomalies based on a PMU, Power and Address tokens consumed within the previous blob. If no anomalies are found, the current state is updated according to the golden model. This anomaly check involves a majority vote from the three streams. A token consumed within a blob may agree with the current state set by the golden model or it may disagree. We define a token as goodToken if it agrees with the current state and a badTokens as one that disagrees. For Power and Address streams, the aggregate count of goodTokens and badTokens are used. A higher count of goodTokens within a blob than badTokens casts a vote in favor of the current state. For the single PMU token consumed at blob boundary, agreement with the current state casts a vote in favor of the current state. The Monitor then makes a transition to the next state as determined by the golden model if votes cast in favor of the current

state exceeds 1. Else, it transitions to Anomaly state.

We test our Monitor on a subset of data collected for the 3 streams. We pick 3 datasets (labeled NR12, NR14, NR17) that do not have an anomaly, and 2 datasets (MR-5, MR-9) that involve an anomaly. Figure 22 and Figure 23 show the test results. The control flow of our test program (Stream benchmark) when represented as Blobs, goes from Blob_0 to Blob_5 where Blob_3 has multiple entry/exit points and three internal paths. These are labeled as Blob_3_0, Blob_3_1, Blob_3_2. These form the columns in our results. Each row represents the vote from individual streams. Agree means that particular stream voted in favor of the current state. Disagree is a vote against the current state. The Monitor was tested on data collected from Blob_1 through Blob_5. The control flow of the anomalous version of our test program goes to the end Blob_5 from Blob_1, bypassing Blob_3 and Blob_4. This is done by the CCI coherence fabric as well. For a preselected return address, it inserts another tampered address. Due to the anomaly, the control flow does not encounter a blob boundary from Blob_1 to Blob_5. The Monitor, therefore, sees it as a single blob. Thus the malicious results show a single vote for the two blobs traversed.

Figure 22 results show 3 out of 24 blobs mis-localized leading to an accuracy of 87.5% for normal execution blob level localization. The anomalies are detected at 100% accuracy.

| NR-12 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Blob_1 | Blob_2 | Blob_3_0 | Blob_4 | Blob_3_1 | Blob_4 | Blob_3_2 | Blob_5 |
| Power | agree | agree | agree | agree | agree | agree | disagree | disagree |
| Address | agree | agree | agree | agree | agree | agree | agree | disagree |
| Pmu | agree | agree | agree | agree | agree | agree | agree | agree |

| NR-14 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Blob_1 | Blob_2 | Blob_3_0 | Blob_4 | Blob_3_1 | Blob_4 | Blob_3_2 | Blob_5 |
| Power | agree | agree | agree | agree | agree | agree | disagree | disagree |
| Address | agree | agree | agree | agree | agree | agree | agree | disagree |
| Pmu | agree | agree | agree | agree | agree | agree | agree | agree |

| NR-17 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Blob_1 | Blob_2 | Blob_3_0 | Blob_4 | Blob_3_1 | Blob_4 | Blob_3_2 | Blob_5 |
| Power | agree | agree | agree | agree | agree | agree | disagree | disagree |
| Address | agree | agree | agree | agree | agree | agree | disagree | disagree |
| Pmu | agree | agree | agree | agree | agree | agree | agree | agree |

;

**Figure 22: Results for Blob-level localization (anomaly-free data)**

| MR-5 | | | MR-9 | | |
|---|---|---|---|---|---|
| | Blob_1 | Blob_5 | | Blob_1 | Blob_5 |
| Power | disagree | | Power | disagree | |
| Address | disagree | | Address | disagree | |
| Pmu | disagree | | Pmu | disagree | |

;

**Figure 23: Results for Blob-level localization (anomalous data)**

Anomaly checks are also done along a path. This is a more frequent check than the Blob transition boundary checks. In this approach, tokens received within a Blob grouped into N-token sized windows. Within a single window, the number of goodTokens and badTokens are counted. This is done for every Blob. The mean and standard deviation information of goodToken/badToken counts per window is extracted for each Blob. During testing, the Monitor does a mean +/- K*standard-deviation test to see if the current window passes this test. Failing to pass this test flags an anomaly. Figure 24 shows the results from this approach for Address stream. Numbers shown are the fraction of windows within a Blob that passed the test. Blob_3_2 being a transition blob, has very few tokens and hence is not sufficient to form a full window. Results for that blob are not available at this point.

**TESTING RESULTS**

Window Size = 2500 tokens | DataSet = NR12
Mean +/- 3*Stddev test results per Blob

| | Blob-1 | Blob-2 | Blob-3_0 | Blob-4 | Blob-3_1 | Blob-4 | Blob-3_2 | Blob-5 |
|---|---|---|---|---|---|---|---|---|
| NR-12 | 0.9993381866 | 0.9659318637 | 0.9868908521 | 0.9881031064 | 0.9865675907 | 0.9873754153 | N/A | 0 |
| NR-14 | 0.9993377483 | 0.9659318637 | 0.9846110003 | 0.9874504624 | 0.982295831 | 0.9874421679 | N/A | 0 |
| NR-17 | 0.9993377483 | 0.9659318637 | 0.9846110003 | 0.9874504624 | 0.982295831 | 0.9874421679 | N/A | 0 |

Mean +/- 1*Stddev test results per Blob

| | Blob-1 | Blob-2 | Blob-3_0 | Blob-4 | Blob-3_1 | Blob-4 | Blob-3_2 | Blob-5 |
|---|---|---|---|---|---|---|---|---|
| NR-12 | 0 | 0.9639278557 | 0.7435166714 | 0.9881031064 | 0.74507002 | 0.9873754153 | N/A | 0 |
| NR-14 | 0 | 0.9639278557 | 0.7227130237 | 0.9881031064 | 0.7170564746 | 0.9881031064 | N/A | 0 |
| NR-17 | 0 | 0.9639278557 | 0.7386719863 | 0.9874504624 | 0.7121644774 | 0.9874421679 | N/A | 0 |

**Figure 24: Results for Path-level localization (anomaly-free data)**

## 4.7 Discussion

Although, the assumption was that the decoupling of raw program state data collection and its analysis by the monitor can be decoupled into two steps to simplify this Phase 1 effort, we did not foresee the additional difficulties this decomposition unleashes. Specifically,

1. The synchronization between streams becomes extremely challenging through this decomposition. In a real-time system, synchronization can be built in implicitly or explicitly despite its challenging nature. But in the decomposed phase model, as in this work, explicit crutches in terms of timestamps and markers have to be created.

2. Misaligned raw data has a huge impact on blob/path detection accuracy further highlighting need for good synchronization.

The signals that are intentionally generated within a program - such as address stream and PMU, have higher fidelity than the unintentional side-channel signals such as power.

Based on our observations with the stream benchmark, complex backtracking at the monitor was not needed. But it is possible that as more complex programs with more complex control flow

(blob flow) graphs are considered, more complex heuristics for backtracking may be needed at the monitor.

The accuracy of many channels (such as PMU) is surprisingly high even if only a segment of an execution path is captured (80%+). This offers interesting overhead-accuracy trad-offs opportunities.

For a completely decoupled monitor, using only the power stream, 70-80% accuracy seems to be achievable.

# 5 Conclusions

Based on the promising results in this effort with the decoupled data collection and monitoring activities, we believe that a real-time data collection, analysis, and decision making prototype should be built.

A real-time prototype unifying data collection, analysis, and decision making will pose challenges in the real time aspects. The analysis through machine learning will be challenging based on the period of the analysis. In our current implementation, the address stream analysis occurs every time an LLC miss address is found. The average separation between LLC addresses seems to be 200-300 clock cycles. Building an SVM engine with 200-300 processor clock cycles latency is feasible. It will involve various trade-offs between feature vector size, efficiency of SVM classification, and the processor clock frequency. We believe that better synchronization afforded by a real-time system can boost the accuracy. This may be offset by the real time considerations (reduced feature vector size in order to perform ML classification in 200-300 clock cycles) to bring back the overall accuracy in the 80-90% range.

Two spaces exist for this kind of anomaly monitor - one where the execution design and program source code are within the end-user control, and the other where the execution engine and the program are black boxes. The former design point exemplified by an SoC with in-house embedded transactional programs can be further explored to include more high-fidelity event channels such as interrupts. The later design point will rely on the power side-channel as the primary indicator.

# 6.0 References

[1] ARM. Arm cortex-a53 mpcore processor technical reference manual.

[2] LLVM. The llvm compiler infrastructure.

[3] Xilinx. Xilinx zcu 106 evaluation board user guide, UG1244 (v1.4) October 23, 2019.

# LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

| | |
|---|---|
| PMU | Performance Monitoring Unit |
| CFI | Control Flow Integrity |
| LLC | Last Level Cache |
| DDR | Double Data Rate |
| FPGA | Field Programmable Gate Array |
| CFG | Control Flow Graph |
| BFG | Blob Flow Graph |
| LLVM | Low Level Virtual Machine |
| COTS | Commercial Off The Shelf |
| PL | Programmable Logic |
| CCI | Cache Coherence Interface |
| L1D | Level-1 Data Cache |
| L1I | Level-1 Instruction Cache |