Enhanced
Instruction Set Randomization Design Space Exploration

M. Tarek Ibn Ziad Columbia University mtarek@cs.columbia.edu
Simha Sethumadhavan, Columbia University, simha@cs.columbia.edu

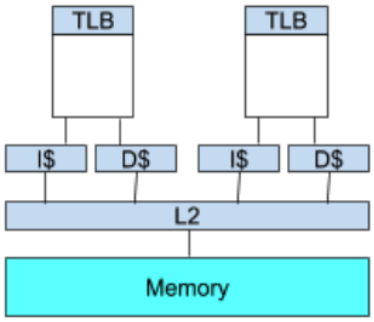
Instruction Set Randomization:

Instruction set randomization (ISR) was proposed early in the last decade as a countermeasure against code injection attacks. It provides illusion of a secret instruction set. However, prior ISR schemes are ineffective against code-reuse attacks. In our previous work, Polyglot [1], we presented the design of a hardware-based ISR scheme, which is effective against code-reuse attacks, and even counter state-of-the-art variants, such as "just-in-time" ROP (JIT-ROP).

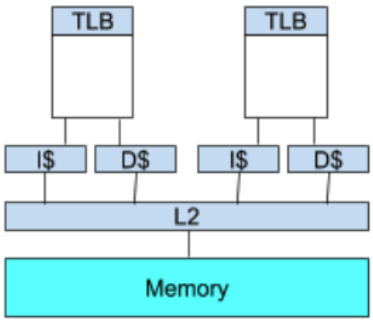
Polyglot creates an "ISRized" binary by symmetrically encrypting (with AES) a diversified version of it, at page granularity, with randomly generated keys. These key-to-address mappings are then asymmetrically encrypted (with ECC) using the target processor's public key and packaged into the binary itself. Since code is encrypted at a page granularity, the executable, and its required shared libraries, possibly encrypted by different sources, are able to interoperate. Lastly, asymmetric encryption ties the binaries to their respective hosts.

To accommodate per-page encryption, Polyglot introduces a new type of page table entry for randomized (i.e., ISR-encrypted) pages. On an instruction page fault, the page walk mechanism procures encrypted entry, decrypts it to obtain the page key and translation, which are then deposited into a modified ITLB. ECC-163 and SHA-256 accelerators are added to the MMU to carry out the decryption according to the Elliptic Curve Integrated Encrypted Scheme. On an I-cache miss, as instructions are fetched from memory, they are decrypted using the page's key and stored, in plaintext, in the I-cache. Henceforth, as long as an instruction is not evicted, execution uses its decrypted form. Moreover, Polyglot employs code randomization to prevent predictable code layout and hence code reuse attacks with low performance overheads. In this project, we analyzed how Polyglot can be extended for Heterogeneous System Architectures (HSA).

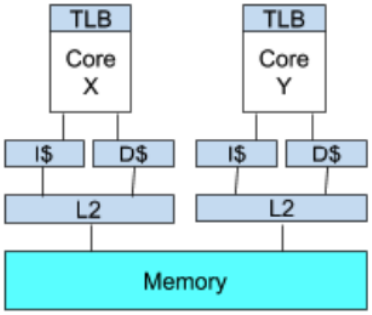
2. Heterogeneous System Architectures (HSA):
HSA may include (1) multicore processors with same ISA and microarchitecture, (2) multicore processors with same ISA and different microarchitecture (e.g., arm big.LITTLE SoC), (3) multicore processors with different ISA and shared virtual memory (e.g., CPU with GPU), or (4) multicore processors with different ISA and disjoint virtual memory (e.g., CPU with discrete GPU). The above categories can have one of the memory sharing schemes shown below.



Same ISA, Same Microarchitecture, Multicore (Shared L2 only)
Ex: **multi-core processor**



Different ISA, Shared Virtual Memory, Multicore (Shared L2 only) Ex: **CPU with GPU (HSA Standards complaint)**



Different ISA, Disjoint Virtual Memory, Multicore (No Sharing)
Ex: **CPU with Discrete GPU**

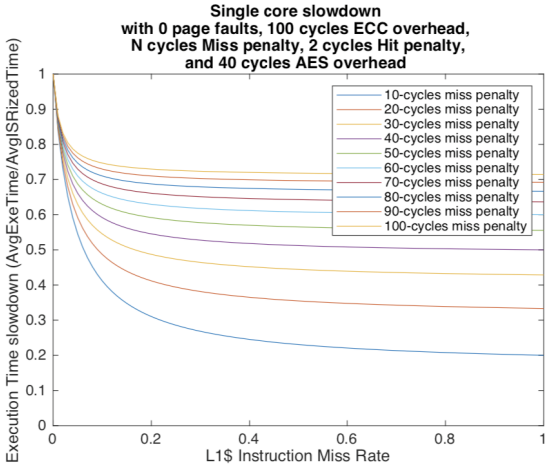
Our goal is to add ISR support to HSA so that no core can read the binary instructions in plaintext form (other than the normal instruction sequence during execution). The performance overheads for handling instruction cache misses and page faults should be minimal.

3. Analytical Evaluation:

We analytically evaluate the overheads of ISR in single and many core environment. First, we provide results for AES decryption overheads (assuming zero page faults). Then, we provide results for handling page faults.

3.A. Analyzing Instruction-Cache Miss Overheads with ISR:

First, we show single core performance slowdown versus L1 I$ miss rate for different miss penalty (to cover a wide range from L2 to L3 or main memory) with a hit penalty of 2 cycles, miss penalty of 60 cycles, and an AES overhead of 40 cycles.



We use the following modeling equations:

$$AvgExeTime = (H * (1 - CM) + M * (CM))$$

$$AvgISRizedTime = (H * (1 - CM) + (M + S) * (CM))$$

$$SlowDown = AvgExeTime./AvgISRizedTime$$

where CM is the percentage of cache miss in a program; it changes with workload [0 < CM < 1], M is the number of cycles for cache miss, H is the number of cycles for cache hit, and S is the number of cycles for AES upon cache miss.

What if we have more than one core? In this case, we may need to handle multiple simultaneous L1 I$ misses. For simplicity, let us consider the case when we have only 1 AES unit. In this case, if we have 2 simultaneous misses, the serialization delay (S) would be S for the first core and 2S for the second core. Similarly, if we have 3 simultaneous misses, the serialization delay (S) would be S for the first core, 2S for the second core, and 3S for the third core. The amount of delay is a function of the simultaneous misses and the available AES cores.

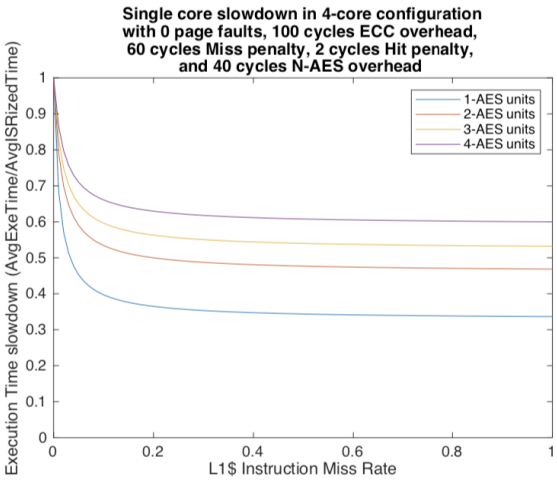
We use random uniform distribution to model the probabilities of having 1,2,3, or n L1 I$ misses at the same time. Then, we calculate the average miss penalty based on those probabilities (weighted the typical L1 miss penalty, 60 cycles).

$$oldMissPenalty = M + S \; for \; single \; core$$

$$newMissPenalty = PrK.* (M + ceil(K/AES) * S)$$

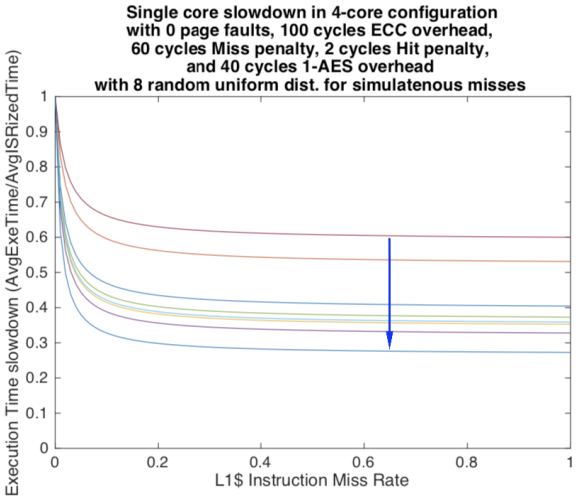
where K is the number of simultaneous misses and PrK is its probability. $AvgISRizedTime = (H * (1 - CM) + (sum(newMissPenalty)) * (CM))$

We show single core performance slowdown in a 4-core configuration versus L1 I$ miss rate for different number of AES units with a fixed miss penalty of 60 cycles and a fixed hit penalty of 2 cycles and an AES overhead of 40 cycles.



We also evaluate the effect of the random uniform distribution of having 1,2,3, or n L1 I$ misses at the same time. The given figure shows 8-different random uniform distributions for 4-cores configuration. It is worth noting that the

figure includes the upper and lower limits; having only n simultaneous misses forever and having only 1 miss at a time.



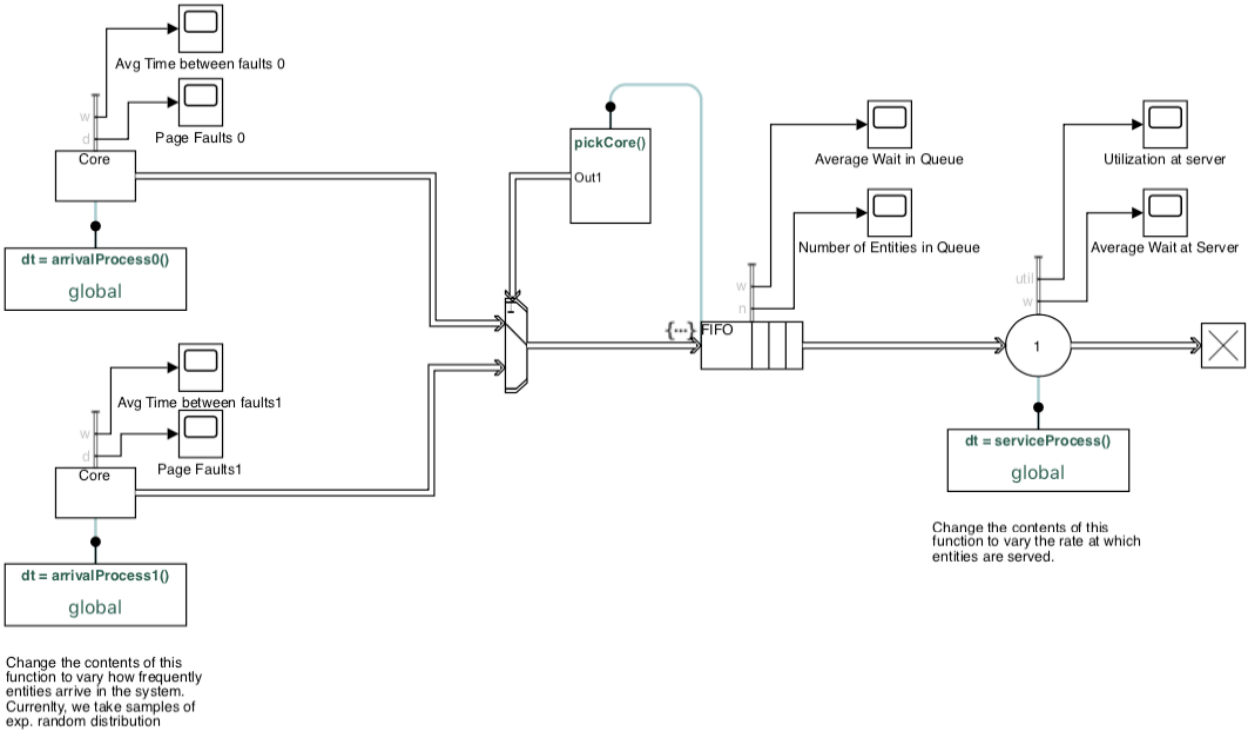
Based on the above analysis, we recommend adding one AES unit close to each core for handling the symmetric key encryption without introducing significant performance overheads.

3.B. Analyzing Page Fault Overheads with ISR:

Here. we focus on analyzing the ISR-ized effect on page faults in many-core environment.

The high level idea is to have n-different cores, with each core generating page faults in a certain manner. We currently use a predefined random distribution. Then, those n-generated page faults would pass through another random probability distribution to simulate the effect of having one or more cores issuing page faults at a time. Finally, the faults reach a queue to be serviced. So, the problem can be modeled as a queueing theory problem and can be solved using queueing theory analysis.

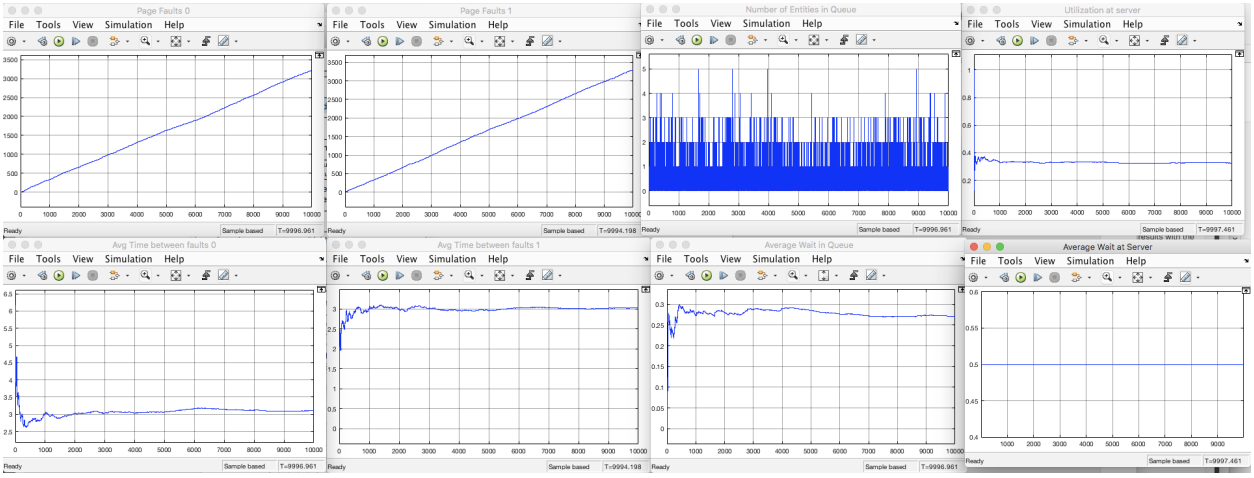
The below figure shows the created model on MATLAB Simulink. Here, we have two cores (shown on the left-hand-side) with their associated global function (`arrivalProcess`) to simulate the effect of generating page faults. Each core has two displays for showing the number of generated faults and the average time between every two successive faults per core.



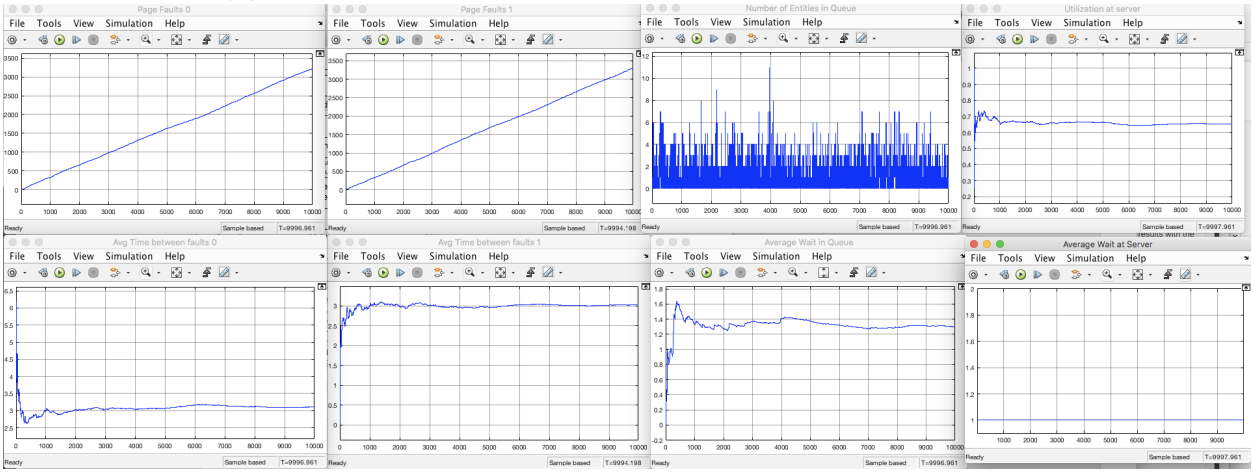
The `arrivalProcess` function is currently configured to generate sample based on an Exponential Distribution with mean=0.5 for both cores. Then, we use a switch, which is controlled by a `pickCore()` function. The switch simulates the effect of picking which core to send the page fault. In the middle of the figure, we have the queue modeled as a FIFO structure with an infinite size. The queue has two corresponding displays; the average wait in the queue and the number of entities (page faults) in queue over time. Finally, we have the server, which represent the page fault handling and ECC decryption overheads. The server has a configurable capacity of one and a function to determine the rate at which faults are handled. We currently set this function to a fixed interval of one simulation tick for page-walks and five

simulation ticks for ECC. The server has two displays; one for the average waiting time (currently useless as we use a deterministic function and not a probabilistic model like with the cores), and a utilization display.

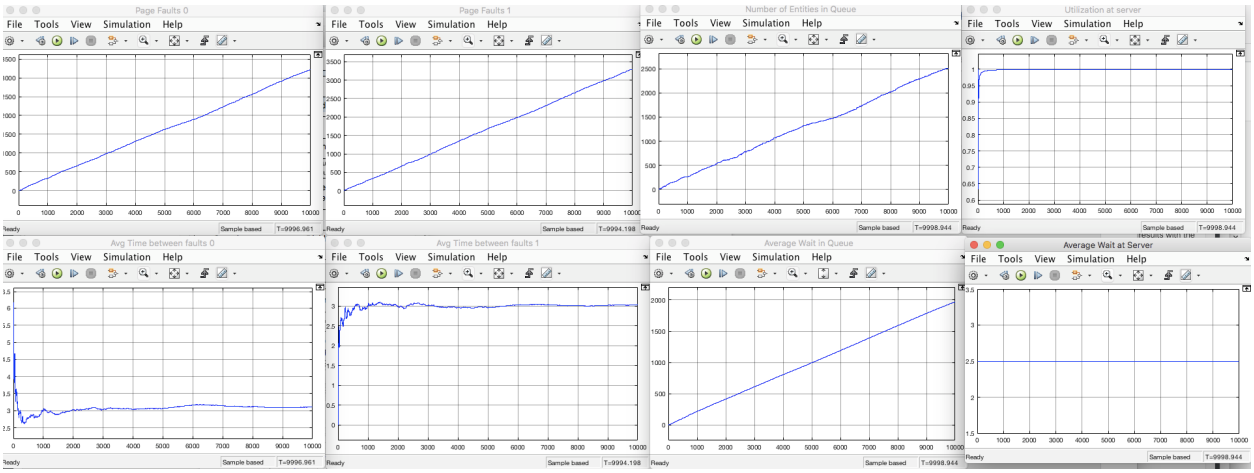
We show the results of a total simulation time of (10,000 simulation ticks). The 2 cores generate page faults with an exponential distribution of 0.5 as a mean. The multiplexer is controlled by a `pickCore` function that uses a random integer (either 1 or 2). The server has a capacity of 1 and the queue has an infinite capacity. First, we simulate the baseline server with a page walk rate of 0.5 (handle 2 page faults per simulation tick) and an ECC rate of 0. We notice an average queuing time of 0.3 with the number of entities in the queue has a maximum of 5 at any given time. The server utilization is 40%.



Second, we simulate the ISR server with a page walk rate of 0.5 (handle 2 page faults per simulation tick) and an ECC rate of 0.5 (same rate). We notice an average queuing time of 1.4 with the number of entities in the queue has a maximum of 11 at any given time. The server utilization is 70%.



Third, we simulate the baseline server with a page walk rate of 0.5 (handle 2 page faults per simulation tick) and an ECC rate of 2 (4 times slower than page walk). This means the server now has a rate slower than the arrival rate of requests to the queue. We notice the average queuing time is linearly increasing over time. The same occurs for the number of entities in the queue. The server utilization is 100% almost all over the simulation time.



The above results suggests using a single pipelined ECC unit for handling the page faults of different cores.

4. Conclusion:

In this work, we investigated the feasibility of applying ISR to Heterogeneous System Architectures. We focus on analyzing the performance overheads associated with handling the instruction cache misses and page faults as they are the two procedures affected by our ISR defense. The estimated area overhead is assumes dedicated cryptographic blocks. However, the availability of cryptographic accelerators on the baseline SoC would help reducing such overheads as we can reuse them for ISR depending upon the underlying SoC.

References:

[1] K. Sinha, V. P. Kemerlis and S. Sethumadhavan, "Reviving instruction set randomization," *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, McLean, VA, 2017, pp. 21-28.