Master's Thesis
(Course Code: XM_0123)

My Thesis: what, why, and how?

by

Yuxuan Chen
(Student Number: 007)

Submitted in partial fulfillment of the requirements
for the degree of
'Computer Security' Master of Science
in
Computer Science
at the
Vrije Universiteit Amsterdam

February 10, 2025

Certified by .....
First supervisor's name
First supervisor's title
First Supervisor

Certified by .....
Daily supervisor's name
Daily supervisor's title
Daily Supervisor

Certified by .....
Second reader's name
Second reader's title
Second Reader

My Thesis: what, why, and how?

Yuxuan Chen
Vrije Universiteit Amsterdam
Amsterdam, NL
anonpenguin@student.vu.nl

ABSTRACT

1 INTRODUCTION

2 BACKGROUND

2.1 Runahead execution

Runahead execution is a speculative execution technique used in modern processors to improve memory-level parallelism. When a long-latency memory operation stalls the pipeline (e.g., due to a cache miss), the processor enters runahead mode. In this mode, it speculatively executes instructions beyond the stall to prefetch useful data into the cache [8].

2.1.1 Runahead vs architectural execution. Architectural execution refers to the updates of the machine state that are visible to the programmer, such as changes to registers and memory writes. It does not involve the specifics of how instructions are executed internally by the processor. For example, an add instruction, at the microarchitectural level, involves several steps: the instruction is first fetched from memory or cache into the processor, decoded by the decoder, and the decoded signals are sent to the control logic. The control logic then directs the relevant register values to the Arithmetic Logic Unit (ALU) for computation, and the result is written back to the register file. However, from the programmer's perspective, only the final update to the register file is observable, and the intermediate microarchitectural processes remain hidden [5].

Runahead operates at the microarchitectural level, meaning that during runahead execution, the CPU's architectural register file is not updated. It is an internal optimization process within the processor and remains invisible to the programmer, ensuring no changes are made to the programmer-visible machine state.

2.1.2 Runahead vs speculation execution. Runahead execution and speculative execution differ primarily in their purpose and impact. Runahead is a microarchitectural optimization designed to improve memory-level parallelism by prefetching data during long-latency stalls, such as cache misses. It does not update the architectural state, as all results produced during runahead mode are discarded, with only prefetched data retained in the cache. In contrast, speculative execution aims to enhance instruction-level parallelism by predicting future instructions, such as branches. If the prediction is correct, the results are committed to the architectural state; otherwise, they are discarded.

2.1.3 Runahead vs prefetching. Prefetching and runahead execution both aim to reduce memory latency but operate differently. Prefetching relies on the analysis of memory access patterns to predict future data accesses and proactively load the data into the cache, often using hardware or software-based algorithms [9]. In contrast, runahead execution handles long-latency stalls, such as cache misses, by speculatively executing instructions beyond the stalled point. Although runahead also triggers memory loads, it follows the program's actual control flow rather than relying solely on predicted patterns. Additionally, runahead does not update the architectural state, as all results are discarded after exiting runahead mode, whereas prefetching directly updates the cache to make predicted data available for subsequent instructions.

2.2 Dynamic code optimization

Nvidia's Denver CPU incorporates a dynamic code optimizer (DCO), which serves as a runtime system designed to improve program performance by analyzing and optimizing code during execution [3]. The overall workflow of the DCO system is illustrated in Figure 1. The branch unit monitors the execution frequency of the basic blocks. When the execution count of a block exceeds a predefined threshold, the CPU forwards the block to the dynamic code optimizer. The optimizer then analyzes the block, translates it into optimized microcode, and directly delivers this optimized code to the execution units, bypassing the standard ARM instruction decoder.

However, this form of runtime optimization presents challenges in analyzing the behavior of the program. By dynamically altering code execution, DCO complicates both program debugging and the study of processor behavior, as described in previous work [6]. The DCO system performs several advanced optimizations, including loop unrolling, loop-invariant loading lifting, reordering of loads and stores, removing redundant instructions, etc.[2, 3].

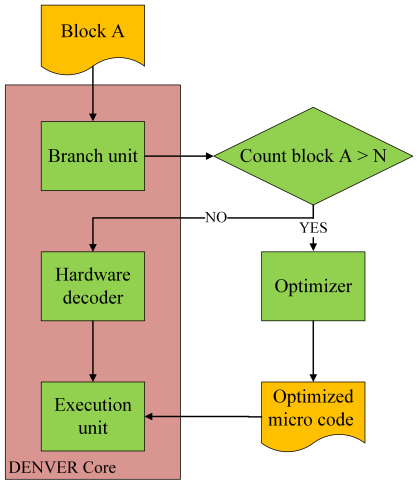

Figure 1: ***Dynamic code optimizer in Nvidia Denver***

3 DCO ANALYSIS

Since the DCO modifies the original code layout during its optimization process, it introduces significant challenges to reverse engineering the processor behavior. These challenges arise because the transformations obscure the original program flow and make it difficult to map the optimized microcode back to the original instructions. Given that the specific implementation details of the DCO are not publicly disclosed, and extracting the underlying microcode from hardware such as the Jetson board is infeasible due to both technical and proprietary constraints, we instead focus on analyzing the observable behaviors of the DCO. This approach allows us to study its impact indirectly.

3.1 Monitor whether the DCO is activated

As discussed in the previous section, the DCO may eliminate unused instructions. Using this feature, we can design an experiment to observe and verify whether the DCO is triggered.

3.1.1 Experiment design and hypothesis. We can insert redundant instructions that are likely to be recognized by DCO as unused code and then measure the latency associated with the execution of these instructions. By analyzing the latency, we can infer whether the DCO is actively optimizing the code and removing these redundant instructions. As demonstrated in algorithm 1, the experiment involves measuring the latency of a bundle of redundant instructions in multiple iterations. For this experiment, we used the "mov reg, reg" instruction as an example of an unused instruction that does not affect the architectural state of the program. The goal is to observe whether the DCO optimizes away these redundant instructions and to monitor the latency to determine if the DCO is triggered during execution. If DCO successfully eliminates these instructions, we expect to see a reduction in latency, indicating that DCO has performed its optimization. In the experiments, we measured the latency of 100 repeated "mov reg, reg" instructions over 10000 iterations to observe the behavior of the DCO.

In this approach, we hypothesize that if the DCO recognizes these instructions as redundant or unused, it may choose to eliminate them from the fetched instructions. If these redundant instructions are removed by the DCO, we will observe a significant reduction in latency after several iterations. This reduction indicates that the DCO has successfully analyzed the code and eliminated these unnecessary instructions from the generated microcode, leading to a more efficient execution. By monitoring this behavior over multiple iterations, we can infer that the DCO has completed its optimization process, as the observed latency will decrease once the redundant operations are no longer executed.

Algorithm 1 Measure redundant instructions in a loop

```
x ← N
while x ≠ 0 do
    time ← counter                ▷ Read performance counter
     execute a bundle of redundant instructions
    time ← counter − time
    x ← x − 1
end while
```

3.1.2 Result and analysis. As shown in Figure 1, a significant latency drop was observed between the 5000th and 6000th iterations. The latency decreased from approximately 180 to 50 cycles, indicating an optimization event. This drop of around 110 cycles corresponds closely to the number of redundant instructions being processed.

The observed latency reduction suggests that the DCO identifies the repeated instructions as redundant, optimizes the execution by removing them, and subsequently generates microcode that significantly reduces execution time. This behavior highlights the DCO's role in improving program efficiency by eliminating unnecessary instructions and tailoring the code layout dynamically based on execution patterns.

In addition to the sudden observed drop in latency, we identified three distinct phases across the 10,000 loop iterations. The first phase (marked in blue) exhibits relatively stable latency, the second phase (marked in green) shows a slight increase in latency, and the final phase (marked in red) demonstrates a significant drop in latency. We hypothesize that, during the first phase, the Dynamic Code Optimization (DCO) has not yet been triggered. In the second phase, DCO is actively analyzing the code, while in the third phase, the CPU executes the optimized microcode generated by DCO. We also observed that in phase 3, there are no further changes, regardless of the number of "mov" instructions. The final latency stabilizes at 50 cycles. We believe that phase 3 represents the stage after DCO optimization, where the processor no longer uses the ARM decoder but directly executes the optimized microcode generated in the translator cache.

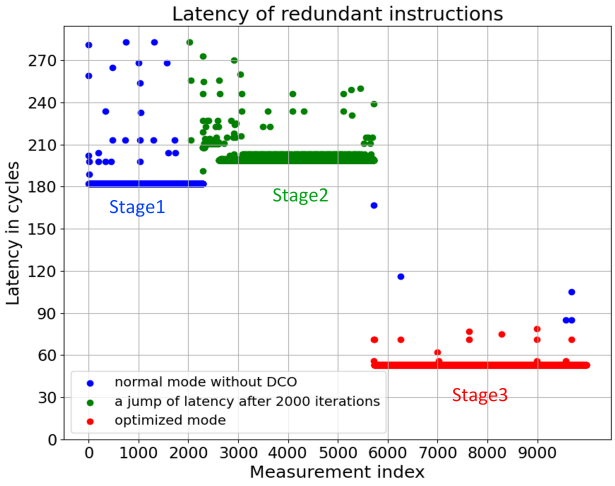

Figure 2: ***Latency measurement for redundant instructions***

3.2 DCO behavior analysis phrase 1

In the experiments observing the triggering of the DCO, we identified three distinct phases in latency measurements for redundant instructions 1: latency stabilizing at 180 cycles, a slight increase at

210 cycles, and finally a significant drop to 50 cycles. These patterns were consistently observed over multiple experimental repetitions.

We believe that in the first phase, the latency remains stable because the processor operates in a normal mode and has not yet reached the threshold to activate the Dynamic Code Optimization (DCO). The number of loop iterations in this phase approximately corresponds to the threshold required to trigger DCO. Moreover, the duration of this phase does not change with variations in the number of instructions within the basic block.

3.2.1 Experiment design and hypothesis. We continue to measure the latency for 100, 200, 300, and 400 mov instructions, each experimental group being repeated 2000 times. We extract the initial region corresponding to the first phase before the latency rise occurs and record the number of iterations that this phase lasts. If the number of iterations in the first phase is similar across these four groups, it suggests that the previously identified first phase corresponds to the latency of instructions executed through the ARM decoder. Moreover, the duration of this phase is approximately equal to the threshold required to trigger the DCO.

3.2.2 Results. Since the iteration count of phase 1 is not stable, we cannot definitively determine the exact threshold value. However, as observed in the box plot 3, the means and data distributions of the four experiments are approximately similar. We can conclude that during phase 1, the processor operates using the ARM decoder, and the threshold is approximately around 2100 iterations.

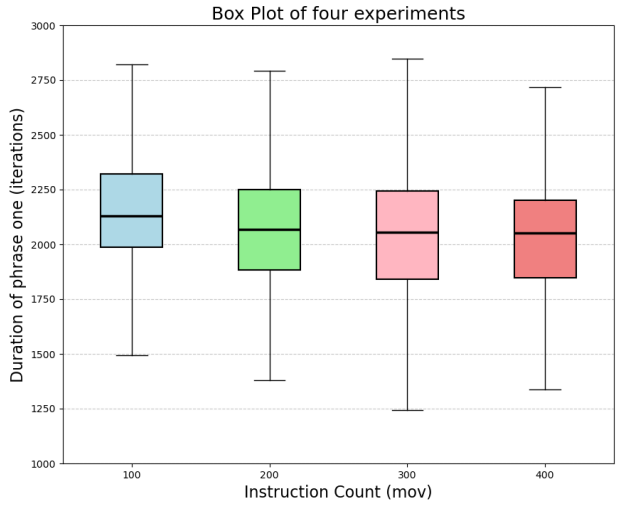

Figure 3: ***Box plot of four experiments which record the duration for phrase one in iterations***

3.3 DCO behavior analysis phrase 2

In the second phase, the latency of the mov instruction block increases by approximately 10-15 cycles. We hypothesize that this delay is caused by the DCO analysis process. However, under idle system conditions, the DCO analysis phase should be handled by another CPU core [2], and such delays should not occur. Therefore, we speculate that this slight increase in latency is caused by interactions between the core running the program and the DCO module.

3.3.1 Experiment design and hypothesis. We continue to use redundant "mov" instructions and record the run-time latency for 100, 200, 300, and 400 "mov" instructions. Subsequently, we extract the duration, in terms of loop iteration counts, for the section where the latency exhibits a slight increase. Since a greater number of instructions should require more time for the DCO to optimize, if the region of slightly increased latency is indeed related to DCO optimization, the duration of this region should progressively increase for 100, 200, 300, and 400 instructions.

If we observe in the experiment that the delay slightly increases as the number of loop iterations increases for 100, 200, 300, and 400 "mov" instructions, we can infer that the region of slightly increased delay observed in previous experiments is related to the optimization of the DCO.

3.3.2 Result and discussion. From Figure 4, it can be observed that the number of loop iterations progressively increases as the number of mov instructions increases from 100 to 200, 300, and then to 400, accompanied by a slight increase in delay. Although the exact value of this growth cannot be determined due to noise factors, the data distribution from the four sets of results suggests that the increase is relatively stable. For every additional 100 mov instructions, the number of iterations for Phrase 2 consistently grows by approximately 600 to 1000 iterations.

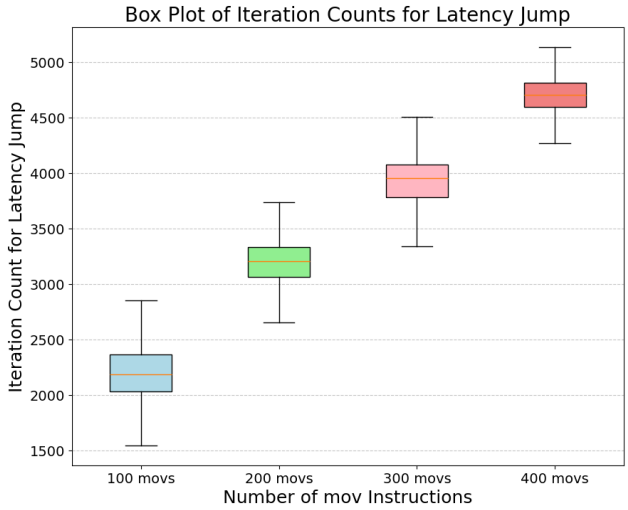

Figure 4: ***Recording the duration of the slight latency increase in terms of the number of loop iterations across four experiments.***

However, according to NVIDIA Denver documentation [2], the DCO analysis process is carried out by another CPU core. Theoretically, the core that executes the instructions should not experience a noticeable increase in delay.

We speculate that the delay is caused by the processor performing a lookup in the translator cache for the corresponding optimized instructions. Specifically, when the number of instructions in a basic block exceeds a threshold, the DCO is triggered and starts optimizing the basic block. Before the final optimized microcode is generated, the processor will perform a lookup in the translator cache at the beginning of the executing basic block to check if the corresponding optimized microcode is available [7].

3.4 Reusing Optimized Code in Subsequent Runs

In our experiments, we observed that the optimized DCO microcode may be retained. If a program undergoes optimization and is not recompiled or modified, it will enter the DCO mode more quickly upon subsequent executions.

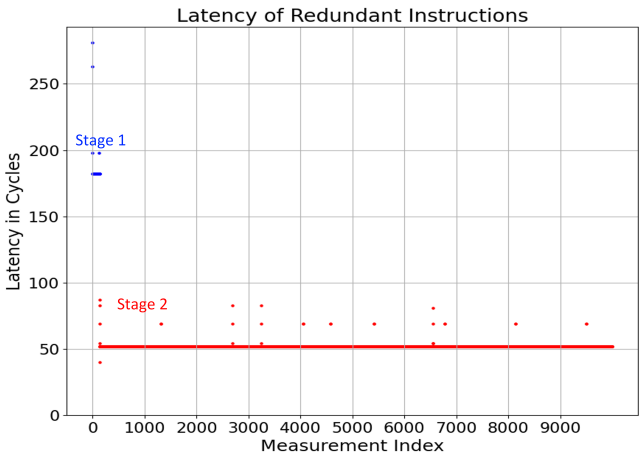

Figure 5: *Optimized Code is reused in subsequent runs*

In figure 5, we can observe that the portion of the instructions decoded by the ARM decoder lasted for approximately 100 cycle iterations. After that, the latency for 100 instructions quickly decreased. Unlike in previous experiments, Phrase 2 is absent, meaning that when the original binary continues to execute, the processor does not go through the DCO analysis phase. Instead, it uses the microcode already stored in the cache.

If the binary files are only deleted and recompiled before each run, there is still a significant chance that the processor will utilize the previously generated microcode. In the experiments we conducted, about 150 out of 2000 runs showed that the latency for the "mov" instructions block decreased after fewer iterations. Therefore, simply recompiling the code does not fully resolve this issue.

3.4.1 Experiments and hypothesis for countermeasure. Each time the number of executions for a basic block exceeds a threshold, the processor will pass the address of this basic block to the binary translator and then perform a lookup in the translator cache based on this address [7]. Nvidia DCO might use a similar design. Therefore, we can ensure that the address of the basic block is different each time the program runs. We can achieve this by inserting a varying number of "NOP" instructions before the basic block, ensuring that the number of "NOP" instructions is different each time 2. In this way, the address of the basic block will be different in each run.

Algorithm 2 Measure Redundant Instructions with Preceding NOPs

```
M ← value        ▷ Change the value each time when compiling
NOPs(M)                                  ▷ A block with M NOPs
x ← N
while x ≠ 0 do
    time ← counter                       ▷ Read performance counter
     execute a bundle of redundant instructions
    time ← counter − time
    x ← x − 1
end while
```

In this experiment, we assume that the processor uses the address of the basic block for the lookup, because when the binary file is run repeatedly without modification, the address of the basic block does not change, and thus it continues to use the microcode already present in the translator cache. If, after inserting a piece of code before the basic block each time, the latency of the "mov" instruction block does not exhibit a rapid decrease across multiple experiments, and there is a slight increase in latency followed by a suspected DCO optimization phase (phrase), then our assumption is correct.

3.4.2 Results. In figure 6, we observe that after inserting a dummy "NOP" block before the loop, the results clearly exhibit three distinct phases, as analyzed in the previous section. After relatively few iterations, the sudden latency drop phenomenon no longer appears. This indicates that the microcode optimized by DCO is likely persistent and can be used continuously. Furthermore, it confirms that the processor indeed relies on the basic block address as a reference for microcode region lookup.

3.5 Preventing the CPU from Entering Optimization Mode

We previously concluded that if a basic block contains more instructions, the time required for the DCO analysis will increase. However, the number of instructions does not affect the threshold for triggering the DCO analysis. However, we still need more effective methods to prevent code execution from entering the DCO-optimized phase; therefore, we need to reduce the number of loop iterations.

3.5.1 Loop unrolling. Loop unrolling is a technique that expands the loop body by replicating its operations multiple times, reducing the overhead associated with loop control. Loop unrolling modifies the termination condition to match the increase in the size of the iteration steps and minimizes or removes redundant branch instructions, improving execution efficiency [4].

As mentioned above, the branch unit monitors the execution frequency of each basic block and uses these counts to determine

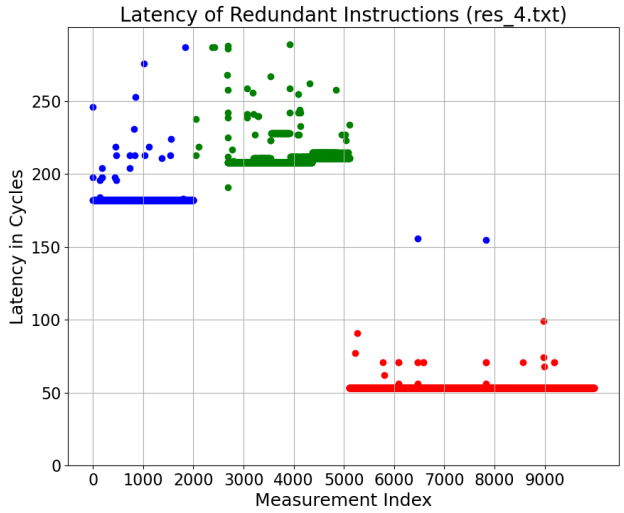

Figure 6: *Latency for mov block after inserting dummy nops*

whether a block qualifies for optimization. However, when techniques such as loop unrolling are applied, the original basic block is altered, leading to an increase in the number of instructions within the block. This can potentially extend the time required for DCO analysis. More importantly, the number of iterations in the loop is significantly reduced, which causes the execution count of the basic block to fall below the threshold required for optimization.

```
/* A simple loop before unrolling */
for(int i = 0; i < 2000; i++){
    x[i] = x[i] + 1;
}

/* Loop unrolling by 4 */
for(int i = 0; i < 2000; i = i + 4){
    x[i] = x[i] + 1;
    x[i + 1] = x[i + 1] + 1;
    x[i + 2] = x[i + 2] + 1;
    x[i + 3] = x[i + 3] + 1;
}
```

Listing 1: Loop unrolling example

3.5.2 Experiment design and hypothesis. We will unroll a loop of 5000 iterations into a single loop of 100 iterations, with 50 delay measurements taken during each iteration. In the loop, we will measure the delay of 100 redundant "MOV" instructions, while also adding 100 dummy "NOP" instructions to increase the number of instructions in the basic block.

If we do not observe a significant reduction in delay (greater than 100 cycles), it indicates that our method is effective.

3.5.3 Result and discussion. From figure 7, we observe that there is no significant reduction in latency. However, in the last 500 to 800 iterations, the latency slightly decreased by about 20 cycles. Since the reduction is minimal, it is likely caused by the processor's conventional optimizations rather than DCO (Dynamic Code Optimization). In addition, on the vertical axis of the graph, we notice some regular and significant fluctuations in latency. We believe these may be due to I-cache misses. Furthermore, as a result of the hardware prefetcher's prefetching behavior, the latency begins to gradually decrease to around 180 cycles.

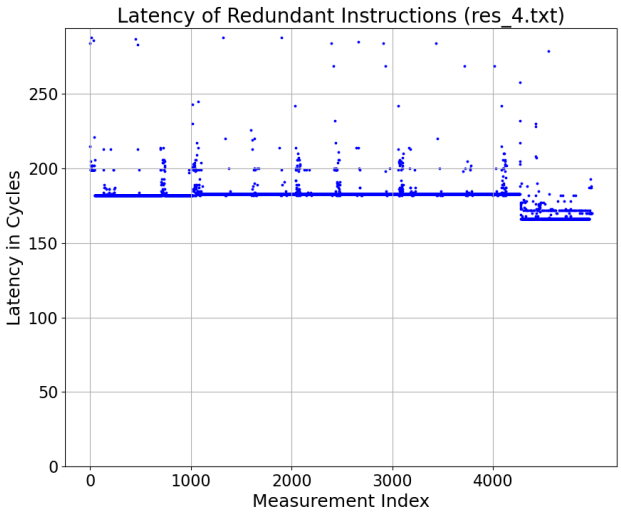

Figure 7: *Latency measurement for redundant instructions with unrolling and inline function*

4 THREAT MODEL

5 OVERVIEW

6 DESIGN

7 EVALUATION

8 DISCUSSION

9 RELATED WORK

10 CONCLUSION

REFERENCES

[1] [n.d.]. Security Research Artifacts. https://secartifacts.github.io.
[2] Darrell Boggs, Gary Brown, Bill Rozas, Nathan Tuck, and KS Venkatraman. 2014. Hot Chips 2014 Nvidia's denver processor. In *2014 IEEE Hot Chips 26 Symposium (HCS)*. IEEE, 1–25.
[3] Darrell Boggs, Gary Brown, Nathan Tuck, and KS Venkatraman. 2015. Denver: Nvidia's first 64-bit ARM processor. *IEEE Micro* 35, 2 (2015), 46–55.
[4] Jack W Davidson and Sanjay Jinturkar. 1995. *An aggressive approach to loop unrolling*. Technical Report. Citeseer.
[5] Joseph E Hollingsworth and Bruce W Weide. 1995. Micro-Architecture vs. Macro-Architecture. In *Proceedings of the Seventh Annual Workshop on Software Reuse*. 101–130.
[6] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. 32–43.
[7] CD James, KG Brian, PB John, J Richard, K Thomas, K Alexander, and M Jim. 2003. The transmeta code morphingTM software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the 1st International Symposium on Code Generation and Optimization. San Francisco, California, USA*. 15–24.
[8] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. IEEE, 129–140.

[9] Steven P Vanderwiel and David J Lilja. 2000. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)* 32, 2 (2000), 174–199.

ARTIFACTS

Based on the standard Artifact Appendix template used by the USENIX Security Symposium. Please follow the self-contained instructions below and assume (i) you are compiling the final version of the Artifact Appendix (no need to cater to reviewers, but to general users of your artifact) and (ii) you are documenting everything in scope for all the artifact evaluation badges (*Artifacts Available*, *Artifacts Functional*, *Results Reproduced*). More information at [1].

A ARTIFACT APPENDIX

This artifact appendix is meant to be a self-contained document which describes a roadmap for the evaluation of your artifact. It should include a clear description of the hardware, software, and configuration requirements. In case your artifact aims to receive the functional or results reproduced badge, it should also include the major claims made by your paper and instructions on how to reproduce each claim through your artifact. Linking the claims of your paper to the artifact is a necessary step that ultimately allows artifact evaluators to reproduce your results.

Please fill all the mandatory sections, keeping their titles and organization but removing the current illustrative content, and remove the optional sections where those do not apply to your artifact.

A.1 Abstract

[Mandatory] Provide a short description of your artifact.

A.2 Description & Requirements

[Mandatory] This section should list all the information necessary to recreate the same experimental setup you have used to run your artifact. Where it applies, the minimal hardware and software requirements to run your artifact. It is also very good practice to list and describe in this section benchmarks where those are part of, or simply have been used to produce results with, your artifact.

A.2.1 Security, privacy, and ethical concerns. [Mandatory] Describe any risk for evaluators while executing your artifact to their machines security, data privacy or others ethical concerns. This is particularly important if destructive steps are taken or security mechanisms are disabled during the execution.

A.2.2 How to access. [Mandatory] Describe here how to access your artifact. If you are applying for the Artifacts Available badge, the archived copy of the artifacts must be accessible via a stable reference or DOI. For this purpose, we recommend Zenodo, but other valid hosting options include institutional and third-party digital repositories (e.g., FigShare, Dryad, Software Heritage, GitHub, or GitLab — not personal webpages). For repositories that can evolve over time (e.g., GitHub), a stable reference to the evaluated version (e.g., a URL pointing to a commit hash or tag) rather than the evolving version reference (e.g., a URL pointing to a mere repository) is required. Note that the stable reference provided at submission time is for the purpose of Artifact Evaluation. Since the artifact can potentially evolve during the evaluation to address feedback from the reviewers, another (potentially different) stable reference will be later collected for the final version of the artifact (to be included here for the camera-ready version).

A.2.3 Hardware dependencies. [Mandatory] Describe any specific hardware features required to evaluate your artifact (CPU/GPU/FPGA, vendor, number of processors/cores, microarchitecture, interconnect, memory, hardware counters, etc). If your artifact requires special hardware, please provide instructions on how to gain access to the hardware. For example, provide private SSH keys to access the machines remotely. Please keep in mind that the anonymity of the reviewers needs to be maintained and you may not collect or request personally identifying information (e.g., email, name, address). [Simply write "None." where this does not apply to your artifact.]

A.2.4 Software dependencies. [Mandatory] Describe any specific OS and software packages required to evaluate your artifact. This is particularly important if you share your source code and it must be compiled or if you rely on some proprietary software that you cannot include in your package. In such a case, you must describe how to obtain and to install all third-party software, data sets, and models. [Simply write "None." where this does not apply to your artifact.]

A.2.5 Benchmarks. [Mandatory] Describe here any data (e.g., datasets, models, workloads, etc.) required by the experiments with this artifact reported in your paper. [Simply write "None." where this does not apply to your artifact.]

A.3 Set-up

[Mandatory] This section should include all the installation and configuration steps required to prepare the environment to be used for the evaluation of your artifact.

A.3.1 Installation. [Mandatory] Instructions to download and install dependencies as well as the main artifact. After these steps the evaluator should be able to run a simple functionality test.

A.3.2 Basic Test. [Mandatory] Instructions to run a simple functionality test. Does not need to run the entire system, but should check that all required software components are used and functioning fine. Please include the expected successful output and any required input parameters.

A.4 Evaluation workflow

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] This section should include all the operational steps and experiments which must be performed to evaluate if your your artifact is functional and to validate your paper's key results and claims. For that purpose, we ask you to use the two following subsections and cross-reference the items therein as explained next.

A.4.1 Major Claims. [Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Enumerate here the major claims (Cx) made in your paper. Follows an example:

(C1): *System_name achieves the same accuracy of the state-of-the-art systems for a task X while saving 2x storage resources. This is proven by the experiment (E1) described in [refer to your paper's sections] whose results are illustrated/reported in [refer to your paper's plots, tables, sections or the sort].*

(C2): *System_name has been used to uncover new bugs in the Y software. This is proven by the experiments (E2) and (E3) in [ibid].*

A.4.2 Experiments. [Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Link explicitly the description of your experiments to the items you have provided in the previous subsection about Major Claims. Please provide your estimates of human- and compute-time for each of the listed experiments (using the suggested hardware/software configuration above). Follows an example:

(E1): *[Optional Name] [30 human-minutes + 1 compute-hour + 5GB disk]: provide a short explanation of the experiment and expected results.*
How to: *Describe thoroughly the steps to perform the experiment and to collect and organize the results as expected from your paper. We encourage you to use the following structure with three main blocks for the description of your experiment.*
Preparation: *Describe in this block the steps required to prepare and configure the environment for this experiment.*
Execution: *Describe in this block the steps to run this experiment.*
Results: *Describe in this block the steps required to collect and interpret the results for this experiment.*

(E2): *[Optional Name] [1 human-hour + 3 compute-hour]: ...*

(E3): *[Optional Name] [1 human-hour + 3 compute-hour]: ...*

In all of the above blocks, please provide indications about the expected outcome for each of the steps (given the suggested hardware/-software configuration above).

A.5 Notes on Reusability

[Optional] This section is meant to optionally share additional information on how to use your artifact beyond the research presented in your paper. In fact, a broader objective of an artifact evaluation is to help you make your research reusable by others.

You can include in this section any sort of instruction that you believe would help others re-use your artifact, like, for example, scaling down/up certain components of your artifact, working on different kinds of input or data-set, customizing the behavior replacing a specific module/algorithm, etc.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.