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

December 3, 2024

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 Dynamic code optimization

Nvidia's Denver CPU incorporates a dynamic code optimizer (DCO), which serves as a runtime system designed to enhance 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 basic blocks. When the execution count of a block surpasses a predefined threshold, the CPU forwards the block to the dynamic code optimizer. The optimizer then analyzes the block, translates it into optimized micro-code, and directly delivers this optimized code to the execution units, bypassing the standard ARM instruction decoder.

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

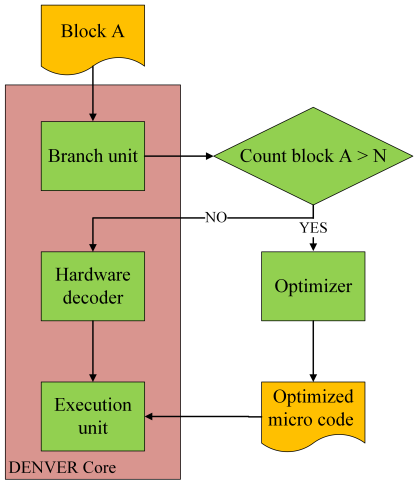

Figure 1: *Dynamic code optimizer in Nvidia Denver*

3 DCO ANALYSIS

Since the DCO (Dynamic Code Optimizer) modifies the original code layout during its optimization process, it introduces significant challenges for reverse engineering the behavior of the processor. 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 Dynamic Code Optimizer (DCO) as unused code and then measure the latency associated with executing 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 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 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.

3.1.2 Result. In the experiments, we measured the latency of 100 repeated "mov reg, reg" instructions over 5000 iterations to observe the behavior of the Dynamic Code Optimizer (DCO). As shown in Figure 1, a significant latency drop was observed between the 3000th and 4000th iterations. The latency decreased from approximately 180 to 70 cycles, indicating an optimization event. This drop of around 110 cycles corresponds closely to the number of redundant instructions being processed.

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
```

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.

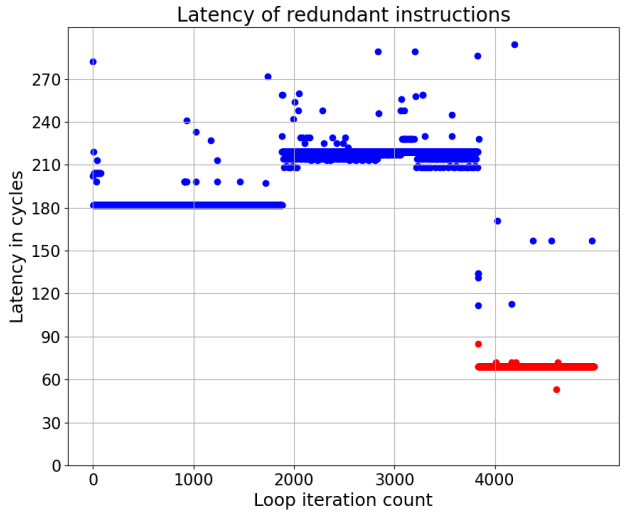

Figure 2: ***Latency measurement for redundant instructions***

3.2 When Does Code Enter the Optimized Mode After DCO Optimization?

Before entering the optimized mode, we hypothesize that our code goes through several stages: ARM normal mode, DCO begins analysis, DCO completes analysis, and then transitions into optimized code mode. We aim to determine whether the number of iterations of our code block is relatively fixed before entering the optimized mode.

3.2.1 Experiment design and hypothesis. Whether the iteration count is below the threshold and DCO has not yet been triggered, or DCO has been triggered but is still in the analysis phase, the processor will operate in ARM normal mode. Since the final microcode has not been generated in either of these two stages, we will not observe a significant reduction in delay until the microcode is generated.

As in the previous experiments, we measure the delay of the redundant code in the loop again, recording the iteration number at which the delay suddenly drops. If a relatively fixed position of delay reduction can be identified, we can estimate the number of iterations spent in the ARM normal mode and the DCO analysis phase (before the optimization microcode is generated) prior to entering the optimization mode.

3.2.2 Result. We placed the "unused instruction" block within a 5000-iteration loop and repeated the experiment 2000 times. Each time, we recorded the iteration point where the delay of the unused instruction block was significantly dropped and stabilized. As shown in the figure 4, the number of iterations required to trigger and complete the DCO analysis is not fixed. However, based on the data-dense regions visible in the figure, the unused instruction block typically experiences a sudden drop in delay after 2000-3000 and 3000-4000 iterations. Among the 2000 experiments, 846 instances occurred in the 2000-3000 iteration range, while 732 instances were observed in the 3000-4000 iteration range.

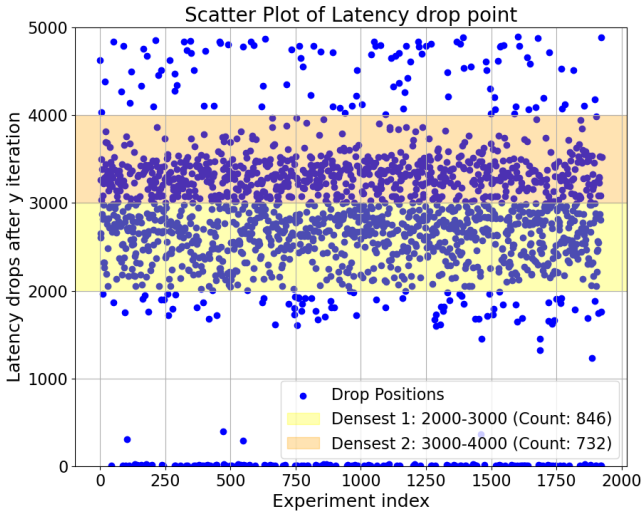

Figure 3: ***Distribution analysis of latency drop positions in DCO across multiple experimental runs***

3.3 Other factors affecting DCO activation

Apart from the execution count of code blocks, we consider additional factors that can influence the triggering and analysis of the DCO.

3.3.1 Experiment design and hypothesis. We hypothesize that the threshold for DCO activation could be relatively lower for function calls, since optimized code within a function can be reused across multiple invocations. In contrast, optimizing basic blocks is less likely to yield reusable code, making such optimizations less efficient. Furthermore, we believe that the size of the instructions within a basic block might impact the time required for DCO analysis and optimization. Although the triggering threshold for DCO

might remain unchanged, larger instruction sets within a basic block could delay the generation of the final microcode.

3.3.2 Result and discussion. In our experiments, we continued to use redundant instructions like "mov reg, reg," placing them within a loop that iterates 5000 times. We then recorded the iteration count at which a significant reduction in latency was observed. For the function-based experiments, we embedded the latency testing code and the redundant instructions within a function and invoked this function within the loop. In addition, we increased the number of instructions to 200 as a comparison to the results with 100 instructions.

From the figure 4, we can observe that for 100 instructions within a function call, a significant portion of the latency reduction points appear in the 2000-3000 iteration range. Moreover, within the 1000-3000 iteration range, the likelihood of DCO triggering and completing its analysis is notably higher compared to the case of 100 instructions directly placed in the loop. However, for the case of 200 instructions directly placed in the loop, the situation is reversed. In this scenario, a significant portion of the latency reduction occurs after 3000 iterations.

From the above experiments, we can infer that the function structure may increase the likelihood of DCO optimization compared to code blocks directly placed in loops. However, in the experiments, the number of redundant instructions within the function was also 100. Theoretically, the time required for DCO to optimize this instruction block should be similar to the basic block scenario in the loop. This suggests that for instructions within a function, the DCO triggering threshold might be relatively lower.

For the case of 200 redundant instructions directly placed in the loop, we observe that entering the DCO execution mode requires more iterations. This leads us to infer that for a larger number of instructions, although the trigger threshold may remain constant, DCO likely requires more time to perform the analysis. Consequently, the total number of iterations needed to transition to the optimized execution mode increases.

3.4 Preventing the CPU from Entering Optimization Mode

From the previous section, we observed that the processor tends to enter the optimized mode when the loop iterations exceed approximately 3000. However, repeated experiments reveal that the exact number of iterations required for the processor to enter the optimized mode is not consistent between different runs. This variation suggests that additional factors, such as system state, cache behavior, or performance variation, could influence the threshold for the processor to enter the optimized mode. As a result, strategies and analysis are required to prevent the processor from entering the optimized mode.

From the analysis above regarding the impact of function calls and instruction count on the time required to enter the optimized mode, we can conclude that, in addition to the most direct factor, the number of loop iterations, function calls may more easily trigger DCO optimization, while a higher number of instructions may increase the time required for DCO analysis. Based on these

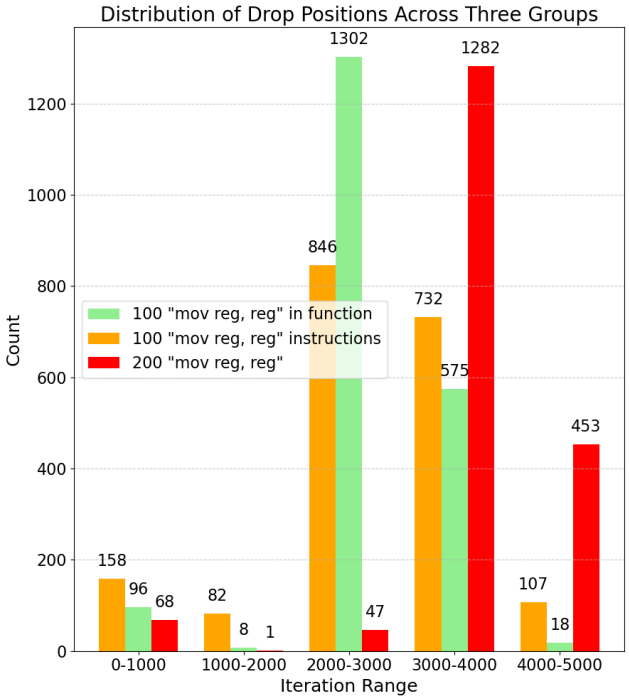

Figure 4: *Distribution analysis of latency drop positions across 100 redundant instructions, 100 redundant instructions in the function, and 200 redundant instructions*

observations, we propose two methods to interfere with DCO optimization, aiming to make the CPU require as many loop iterations as possible before entering the optimized mode.

3.4.1 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. 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 [5].

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

3.4.2 Inline function. Inlining refers to replacing a function call with the actual code of the function, substituting parameters, and resolving identifier conflicts at compile time [4]. Inlining a function removes the computational overhead associated with function calls and returns, enhancing performance by executing the function's code directly within the calling context.

DCO may be more inclined to optimize functions because the threshold for optimization could be lower for function calls compared to regular basic blocks. This hypothesis arises from the observation that the microcode generated for a function could be reused across multiple calls, potentially making the optimization process more efficient. In contrast, for regular basic blocks, the microcode is typically specific to the block and may not have the same reuse potential.

3.5 Evaluation of Existing Methods for Interfering with DCO Optimization

In the previous sections, we hypothesized that loop unrolling and inline functions could interfere with DCO optimization, potentially increasing the time required for the processor to enter the optimization mode. In this section, we will implement these methods in our experiment and observe their effectiveness.

3.5.1 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. Additionally, we will change a timing measurement function for a set of redundant instructions (mov reg, reg) into an inline function to observe if there is a sudden reduction in delay.

If we observe that the number of times a sudden reduction in delay occurs decreases significantly, that would indicate that our method is effective.

3.5.2 Result and discussion. In this experiment, we also ran the program 2000 times. In each run, we performed 50 delay measurements on the redundant instructions in 100 iterations of the loop. We did not observe any delay reduction greater than 50 cycles in any of the runs. The figure 5 shows the delay of 5000 measurements in one experiment. For all experiments, there was no delay reduction greater than 50 cycles.

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.

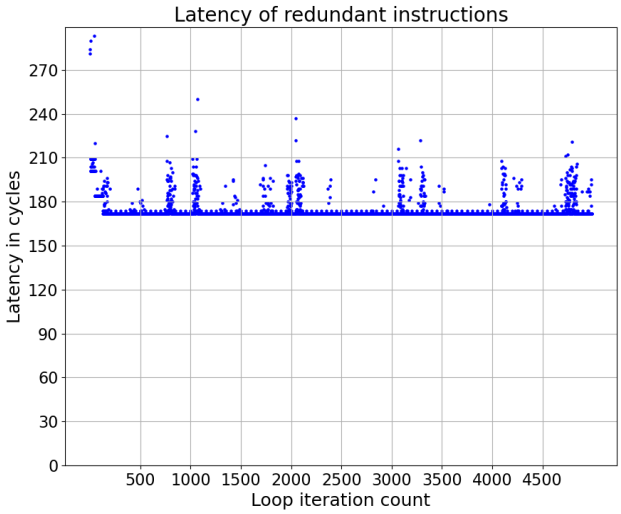

Figure 5: *Latency measurement for redundant instructions with unrolling and inline function*

[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 Anne M Holler. 1988. A study of a C function inliner. *Software: Practice and Experience* 18, 8 (1988), 775–790.
[5] Jack W Davidson and Sanjay Jinturkar. 1995. *An aggressive approach to loop unrolling.* Technical Report. Citeseer.
[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.

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/.