ShuffleV: A Microarchitectural Defense Strategy against Electromagnetic Side-Channel Attacks in Microprocessors

Nuntipat Narkthong*, Yukui Luo†, Xiaolin Xu*
*Northeastern University, Boston, USA †Binghamton University, New York, USA
{narkthong.n, x.xu}@northeastern.edu, yluo11@binghamton.edu

***Abstract*—The run-time electromagnetic (EM) emanation of microprocessors presents a side-channel that leaks the confidentiality of the applications running on them. Many recent works have demonstrated successful attacks leveraging such side-channels to extract the confidentiality of diverse applications, such as the key of cryptographic algorithms and the hyperparameter of neural network models. This paper proposes *ShuffleV*, a microarchitecture defense strategy against EM Side-Channel Attacks (SCAs). ShuffleV adopts the moving target defense (MTD) philosophy, by integrating hardware units to randomly shuffle the execution order of program instructions and optionally insert dummy instructions, to nullify the statistical observation by attackers across repetitive runs. We build ShuffleV on the open-source RISC-V core and provide six design options, to suit different application scenarios. To enable rapid evaluation, we develop a ShuffleV simulator that can help users to (1) simulate the performance overhead for each design option and (2) generate an execution trace to validate the randomness of execution on their workload. We implement ShuffleV on a Xilinx PYNQ-Z2 FPGA and validate its performance with two representative victim applications against EM SCAs, AES encryption, and neural network inference. The experimental results demonstrate that ShuffleV can provide automatic protection for these applications, without any user intervention or software modification.**

***Index Terms*—Side-Channel, Defense, Microarchitecture, RISC-V**

I. Introduction

Modern appliances and everyday objects increasingly rely on embedded microprocessors to execute software, advancing their functionality. However, the run-time electronic characteristics of microprocessors, such as the power trace or EM emanation, present side-channels that can be leveraged to extract the confidentiality of applications. Since the demonstration of a successful side-channel attack (SCA) retrieving the encryption key of the Data Encryption Standard (DES) in [1], [2], many research efforts have been devoted to attack various cryptographic algorithms (e.g., the Advanced Encryption Standard (AES) [3], [4]) and develop countermeasures accordingly.

In recent years, more edge (e.g., smart home and IoT devices) processors have become capable of running high-performance neural network models. In this way, the model inference can be performed locally with high efficiency. Although promising, these emerging systems also create a new attack surface, where an adversary can illegally extract (i.e., using side-channel analysis) the confidentiality of a neural network model. Recent works by Batina *et al.* [5] and Takatoi *et al.* [6] demonstrated successful reverse engineering attacks using side-channels to extract the neural network model architecture and weights from the microprocessors. Since producing a high-performance neural network model demands a large number of labeled data and substantial computational resources, such model architecture and parameters should be treated as the confidentiality of the model owner and well protected. Moreover, acquiring the model architecture and parameters can also aid the attacker in conducting adversarial attacks [7] or deducing training data from the parameters [8]. These emerging attacks raise a new side-channel vulnerability, as the current trend is to reduce latency and power consumption while promoting user privacy by moving inference from the cloud to the edge processors, which makes these systems more susceptible to these side-channel-aided attacks.

Although various countermeasures have been developed for cryptographic encryption standards [9], [10] and neural networks [11]–[14], they have different limitations. For example, software-oriented solutions are easy to deploy in a microprocessor but introduce high performance overhead. On the other hand, hardware-oriented defense techniques are usually targeted customized circuit (i.e., ASIC) and are not applicable to protect general microprocessors. More importantly, most of these existing countermeasures are application-driven e.g., only for securing the encryption key of AES, significantly limiting their applicability. In contrast, microarchitectural-level defense is a promising direction that can automatically safeguard any software executing on a processor without additional efforts from developers or any code modifications.

To embrace these advantages of architectural defense, this paper proposes *ShuffleV*, a microarchitectural defense strategy against EM SCAs. ShuffleV adopts the moving target defense (MTD) philosophy at the architectural level. Specifically, ShuffleV integrates hardware units to randomly shuffle the execution order of program instructions and randomly insert dummy instructions to prevent adversaries from using the statistical information for side-channel analysis. As a result, ShuffleV can provide automatic protection against EM SCAs.

To ensure our findings are reproducible and broadly applicable to real-world systems, we build ShuffleV on the open-source RISC-V core and evaluate its performance on a Xilinx XUP PYNQ-Z2 FPGA board. Specifically, we extend the Ibex RISC-V core [15], an in-order, single-issue core with 2

pipeline stage. Being fully compatible with the original Ibex core interface-wise, ShuffleV can also be used as an drop-in replacement core in the OpenTitan SoC [16] and several SoC in the PULP platforms [17] like PULPissimo, PULPino, and OpenPULP.

We make the following contributions in this work.

- We propose ShuffleV, a side-channel resistant RISC-V core which integrates hardware units to randomize instruction execution order of any program to thwart EM SCAs without developer intervention or software modification.
- We implement ShuffleV on FPGA and evaluate its performance with diverse workloads, including CoreMark benchmark [18], AES-128 encryption [19], and neural network inference on TensorFlow Lite Micro [20]. To the best of our knowledge, this is the first work discussing countermeasures on a general-purpose processor that include performance and security analysis for neural network workload.
- We perform correlation electromagnetic attacks (CEMA) with two representative victim applications, AES and neural network, to evaluate the security performance of ShuffleV. For a fair comparison, we establish baseline references using the well-established open-source RISC-V core Ibex [15] and its security-enhanced version Secure Ibex[1]. The experimental results demonstrate the effectiveness of ShuffleV in securing these two workloads.
- We implement ShuffleV[2] as an drop-in replacement to the open-source Ibex core, which enables our method to be used in many existing SoC design in the PULP and OpenTitan project that is compatible with the Ibex core.
- We equip ShuffleV with 6 design options to suit different design objectives. To facilitate fast performance emulation, we develop a ShuffleV simulator[3] to simulate the performance overhead and estimate the security enhancement on each workload.

II. Background and Related Works

A. Side-channel Attack Overview

Side-channel attack (or SCA) is a type of advanced attack threatening a wide range of systems, including but not limited to cryptosystems. Unlike traditional attacks that exploit the deficiency of software programs, SCA focuses on passively observe, collect, and analyze the information gleaned from the physical components during the system execution, from which the attacker can indirectly divulge information about the victim applications. The representative side-channels include the power trace [1], EM emanation [5], [6], and timing information [21].

[1]We refer to the Ibex core with dummy instruction insertion feature enabled as Secure Ibex in this paper.

[2]ShuffleV core is open-source and available at https://github.com/nuntipat/ShuffleV-Demo-System.

[3]ShuffleV simulator is open-source and available at https://github.com/nuntipat/ShuffleV-Simulator.

Correlation Electromagnetic Attack (CEMA) is a specific variant of SCA that studies the correlation between the EM emanation of a victim device and the data it processes, to extract secret information. CEMA necessitates collecting and statistically analyzing a large set of EM traces from the device under test (DUT). The initial step in this process is to create an appropriate EM model that reflects a certain part of the DUT, i.e., a module like a register that retains intermediate computational values reliant on the secret. Following this, a temporal correlation is identified within the EM trace at when the register updates with the intermediate computational value. The predicted EM values from the model are then cross-referenced with the actual EM traces. Given that the secret within the EM model is unknown, all potential secret values are evaluated through hypothesis testing. The accurate value is the one that yields the highest correlation between the model predictions and the actual measurements.

CEMA has been proven effective in retrieving secrets from microprocessors, such as the cryptographic key of the Advanced Encryption Standard (AES) [22], a long-term victim example in side-channel research. More recently, EM SCAs are also used in reverse engineering the confidentiality of neural network models from microprocessors [5], [6], i.e., to extract the neural network architectures and weights.

B. Defense against SCAs

In recent years, various defenses against Side-Channel Attacks (SCAs) have been proposed. Initially driven by attacks on cryptographic algorithms, most existing defenses are designed to secure implementations of ciphers like AES by protecting sensitive information such as cryptographic keys. Following the successful application of side-channel analysis to reverse-engineer neural network models, several defenses have also emerged to protect Deep Neural Network (DNN) model architectures and weights.

Many defenses have been developed for both hardware and software implementations of cryptographic algorithms. On the software side, a notable work is Rosita [9], a program rewriting engine that automatically protects masked AES implementations based on the input of a leakage emulator. For hardware, Bilgin *et al.* [23], [24] and De Cnudde *et al.* [25] proposed secure hardware implementations based on Threshold Implementations (TI) [26], securing against first-order and second-order power analysis attacks, respectively. Building on this, Gross *et al.* [27] introduced Domain-Oriented Masking (DOM), a technique offering comparable security to TI with reduced chip area and fewer random bits.

For neural network workloads, Liu *et al.* [13] proposed obfuscating memory access patterns by shuffling NN weight memory accesses and adding dummy access signals. Luo *et al.* [14] presented more advanced scheduling obfuscation methods to protect against DNN architecture reverse engineering on FPGA-based DNN accelerators. Beyond shuffling, masking techniques have been explored by Dubey *et al.* [11], [12] to protect linear and nonlinear operations in hardware neural network implementations. Their results indicated an overhead

of 3.5% in latency and a 5.9x increase in area on a Xilinx Spartan-6 FPGA.

In addition to application-specific defenses, another line of work focuses on developing generalized defense strategies to mitigate side channels for diverse applications at the microarchitecture level, using techniques such as shuffling, code morphing, and masking. Bayrak *et al.* [28] utilized a customized hardware unit to randomize the execution order of independent instruction blocks, which are either manually defined by the developer or detected during compile time using their proposed toolchain. Antognazza *et al.* [29] proposed Metis, an integrated hardware module for transparent code morphing at the microarchitecture level, replacing each target instruction with several equivalent instructions. While their approach significantly reduced execution time overhead by 21-141x compared to software-based continuous code morphing, their AES-128 encryption execution time remained 1.8 to 2.35 times slower than that of an unprotected core. Gross *et al.* [30] employed the DOM technique to protect the register file, ALU, and data memory interface of the V-scale RISC-V processor. The evaluation results showed that their design secured the Authenticated Encryption Scheme (ASCON) [31] against first-order attacks, although with a 1.59x increase in LUTs and a 1.84x increase in registers on a Xilinx Spartan-6 FPGA.

ShuffleV falls into this category of generalized defenses. It proposes integrating hardware units to randomize the instruction execution order of any program to thwart EM SCAs without requiring developer intervention or software modification. Our approach improves upon [28] by eliminating the need for source code modification or recompilation with a custom compiler toolchain. Compared to [29], our instruction shuffling approach incurs significantly less execution time overhead because it creates execution randomness by permuting actual program instructions, rather than performing code morphing, which expands one instruction into several to obfuscate the actual computation.

III. Threat Model, Setup, and Baseline

A. Threat Model

This work focuses on microcontrollers and small embedded processors used in embedded and edge IoT devices, which are directly accessible to users and attackers. We take the EM side channel attack as case study, for the following reasons:

- *Single-tenant:* These types of systems run bare-metal software or real-time operating systems (RTOS), such as [32], [33], which contain only trusted manufacturer firmware. Therefore, we can eliminate common attacks in PC, mobile, and cloud environments based on cache, timing, branch predictors, and other microarchitecture side-channels that are launched by side loading malicious software onto the device, such as [21], [34], [35].
- *Location:* These types of system are wildly deployed on the edge and can be easily disassembled and probed. Thus, they are more vulnerable to physical SCAs.
- *Non-invasive:* EM is one of the most feasible and applicable attacks in real-world scenarios, as it doesn't require physical modifications to the printed circuit board (such as removing the capacitor or cutting the power trace), unlike power side-channel attacks. It also doesn't cause irreversible damage to the target device like other invasive attacks, e.g., voltage/clock glitching, and laser injection.

We assume that the attacker has physical access to the device to measure the EM emanation and can send in the input and read the result of the computation from the device.

B. Experimental Setup

Our experimental setup is shown in Fig. 1, which consists of a Xilinx XUP PYNQ-Z2 [36] FPGA board equipped with a ZYNQ XC7Z020-1CLG400C System-on-Chip (SoC). All hardware designs are synthesized using the Xilinx Vivado Design Suite 2023.1 and run at 50 MHz. We employ Tektronix MSO44 4-BW-1000 mixed signal oscilloscope [37], Langer RF-B 0.3-3 H-field EM probe [38], and Langer PA 303 Preamplifier [39] to collect the EM side-channel leakage from the FPGA chip.

We consider two representative workloads: an AES-128 encryption (key = 16 Byte) and a $\sum_{i=1}^{5} in_i \times w_i$ (5i5w) Multiply Accumulator (MAC) operation, where in represents the input data from the user, w is the weight parameter, and i denotes the i^{th} input/weight. Following [5], [6], we select the MAC operation to represent the neural network workload, as it is the core operation that formulates the convolutional and fully connected layers, the core building blocks of the neural network models. To ensure fairness, one million random input samples are pre-generated for each workload and consistently used during EM trace collection across all core configurations.

To perform the attack, we feed each pre-generated input and collect one EM trace per input. The optimal position of the EM probe is determined by continuously running the target application and moving the probe in a grid pattern to find the location that yields the highest signal amplitude at the system's operating frequency. The length of each trace varies depending on the specific workload and the actual execution time of each run, as illustrated in Fig. 2. To simulate a highly capable attacker and evaluate each defense mechanism under a worst-case scenario, a trigger signal is added into the code to indicate the start and stop of computation, enabling synchronized EM measurements. The underlying idea is that if an attack proves unsuccessful under these ideal conditions, its likelihood of success in a real-world environment where misalignment and measurement noise are increased is significantly diminished.

C. Baseline Characterization

Baseline Setup. We use Ibex [15], an open-source RISC-V core, and its enhanced version Secure Ibex [15] as our baseline. The Ibex core is chosen as it is designed specifically for embedded control applications and is equipped with various security features, such as register file error correcting code (ECC), cache ECC, dummy instruction insertion, and data-independent timing to defend against fault injection attacks as well as timing, power, and EM-based side channel leakages. More importantly, the Ibex core is widely adopted as a key

TABLE I: The number of EM traces required to extract AES-128's key on the targeted core configuration.

Config \ Byte#	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
Ibex [15]	300	300	400	150	400	200	100	100	150	250	250	250	1050	100	450	450
Secure Ibex [15]	80k	51k	93k	31k	53k	43k	18k	10k	10k	13k	13k	10k	12k	3k	14k	1k

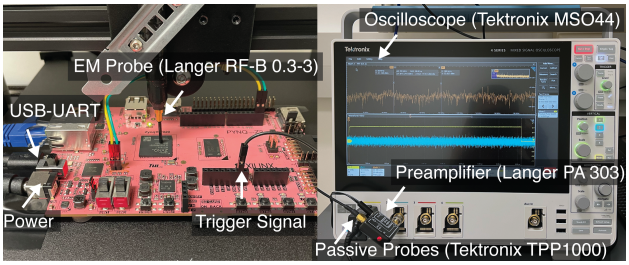

Fig. 1: Our setup for performing EM measurements.

TABLE II: The number of EM traces required to extract the 5i5w MAC weights on the targeted core configuration. *1The attack was performed on weights 1 and 2 simultaneously.

Config \ Weight (i)	1	2	3	4	5
Ibex [15]	270	170	300	150	670
Secure Ibex [15]	37k*1	37k	10k	14k	36k

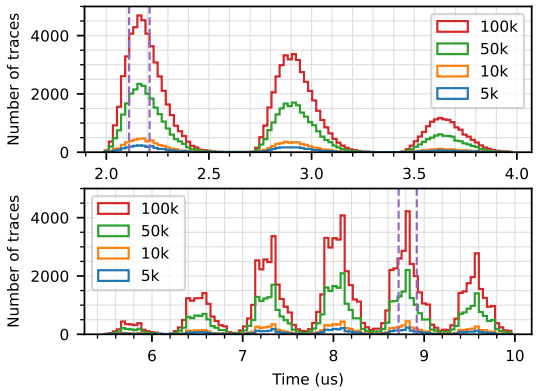

Fig. 2: Histogram of the length of EM traces collected from the 5i5w MAC operation (top) and the AES-128 encryption (bottom) programs run on the Secure Ibex core. Purple dashed lines indicate the minimum and maximum lengths of traces used for the attack.

component in the PULP [17] and OpenTitan [16] platforms and is one of the most popular open-source RISC-V cores on GitHub, with over 1,100 stars. To evaluate these cores, we utilize the Ibex demo system [40], which provides debug support and some necessary peripherals in its default configuration.

The Secure Ibex core is equipped with a dummy instruction insertion mechanism to enhance security by randomly adding one dummy instruction every few actual instructions. We enable this mechanism (by asserting the `dummy_instr_en` control bit in the cpuctrl register) and set the dummy instruction interval (DII) to 16 to insert dummy instructions every 0–16 real instructions. Four types of instruction are used as dummy instructions, each requiring a different number of CPU cycles: `ADD` (1 cycle), `AND` (1 cycle), `MULT` (2 cycles), and `DIV` (37 cycles). These varying cycle counts play a role in *moving target* of the victim applications, e.g., making it more challenging to analyze side-channel traces from repetitive runs.

Attacking Ibex. We first conduct CEMA attacks using the Hamming weight leakage model on the Ibex core [15]. We show the experimental results in the 1^{st} row of Tab. I and II, from which we can observe that, the CEMA attack can easily extract the secret from the Ibex core, i.e., only several hundred traces are needed to extract the AES key and the MAC weights.

Attacking Secure Ibex. Although the Secure Ibex presents enhanced robustness against CEMA, we find it is still possible to conduct a successful attack. We illustrate our attacking idea in Fig. 2, which shows the distribution of program execution time when capturing 5k, 10k, 50k, and 100k EM traces. We can observe that, although the insertion of dummy instructions corrupts the repetitive accumulation of side-channel features, these EM traces can still be clustered into a small number of groups, each following a normal distribution. Therefore, by collecting only traces of comparable length, such as the one between the two purple dashed lines, we can easily gather a substantial quantity of aligned traces. From these experiments, we found that 10k EM traces is sufficient to build an accurate distribution. Following this, an additional 100k traces of comparable length are collected and CEMA is executed to extract the secret from both AES and MAC operation, as shown in the 2^{nd} row of Tab. I and II.

Lessons Learned. Our experimental results in Tab. I and II demonstrate that using a dummy instruction insertion mechanism alone may not provide sufficient protection against EM SCAs. Although Secure Ibex can insert instructions more frequently to enhance security, e.g. every 8 or 4 program instructions, doing so incurs significant performance penalties. Fig. 3 illustrates this overhead by comparing the execution time of the Secure Ibex core to the baseline Ibex core. For the MAC and AES-128 workloads, Secure Ibex's execution time is, on average, 37% and 81% higher, respectively, when DII is set to 16. This increases to 61% and 142% higher when DII is set to 8. Another significant drawback of Secure Ibex is its high execution time variation, which makes it unsuitable for real-time embedded systems that demand predictable timing. To overcome these shortcomings, this paper proposes

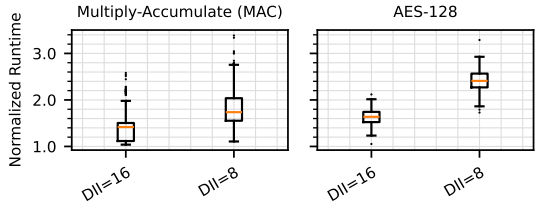

Fig. 3: Normalized execution time of the Secure Ibex with dummy instruction interval (DII) = 8 and 16 compared to the Ibex core on MAC and AES-128 workload (lower is better).

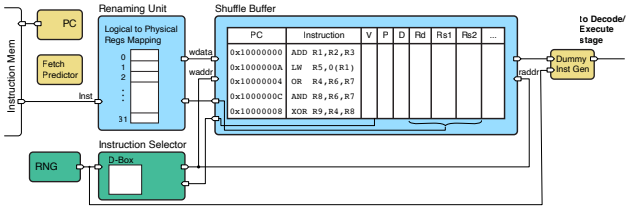

Fig. 4: ShuffleV's high level diagram of the instruction fetch and shuffle process.

ShuffleV. Our method employs a nondeterministic instruction shuffling technique to achieve superior resistance to EM SCAs, significantly reducing execution time overhead to just 13.7% for the MAC workload and 3.1% for the AES-128 workload, all while demonstrating much lower execution time variation.

IV. ShuffleV: Design Overview

To mitigate both security and performance concerns of Secure Ibex, we propose a microarchitectural defense strategy *ShuffleV*, which adopts the *moving-target-defense* (MTD) philosophy. Following our baseline characterization in Sec. III-C, we use the RISC-V ISA specifically the base integer ISA (RV32I) to introduce ShuffleV. Nevertheless, the design is directly applicable to other RISC-V ISA extensions described in [41], e.g., compressed (RVC), multiply and divide (RVM), atomic (RVA), single/double floating point (RVFD), etc., as well as other ISAs such as ARM.

The general idea of ShuffleV can be described as follows: as a MTD-based defense solution, it randomizes the execution sequence of instructions. Particularly, instead of fetching and executing each instruction one-by-one, ShuffleV fetches N instructions to fill a hardware unit, *shuffle buffer*, which stores the fetched instruction awaiting for execution in the next step. Each entry in the instruction buffer contains a program counter (PC) value, the machine instruction, a valid bit, N dependency bits to indicate which other instructions this instruction depends on, and the index of the physical source and destination registers, as illustrated in Fig. 4. Then, ShuffleV randomly selects one pending instruction from the buffer to execute, and refills the buffer with new instructions retrieved from the instruction memory or an instruction cache, if available. Note that the shuffle buffer size is reconfigurable, i.e., can be arbitrarily chosen. While a large buffer size helps to improve execution randomness, it also incurs more hardware and execution time overhead. In our experiments, we found that a buffer size equal to 4 yields a good trade-off between security and hardware overhead (see Sec. VI).

A. Instruction Dependency Tracking

Although the MTD philosophy is straightforward, its implementation is challenged by **how to track data dependency between instructions to maintain program correctness.** In an ideal scenario, there will be N^M ways that a program can be executed, where N is the size of the shuffle buffer and M is the number of CPU cycles required to execute the program, with no performance penalty aside from the larger hardware footprint for the shuffle buffer and the auxiliary logic. In practice, however, not all instructions can be selected for execution in each cycle, due to data dependency between instructions. As a result, the number of unique permutations of the program instructions will be lower than N^M, as shown in Fig. 14. Moreover, control flow statements such as branch or jump also prevent the core from fetching new instructions, while waiting for the branch or jump to execute. Thus, the processor will need to stall for a few cycles after every branch or jump to refill the shuffle buffer, negating the overall performance of the processor, as shown in Fig. 6.

In RISC-V, all instructions perform computation on the register with one destination register and two source registers, except a few instructions in RVF and RVD extensions that have three source operands. The RISC-V's common instruction format and addressing mode make it an ideal candidate for implementing our instruction shuffling strategy. Leveraging this feature, ShuffleV checks whether the next instruction depends on any previous instruction, by comparing the next instruction source register (`rs1`, `rs2`, and `rs3`) with destination registers (`rd`) of all pending instructions. This simple strategy covers most instructions in the RISC-V ISA, except the load/store, synchronization, environment, and control status register instructions, which require special attention.

One critical optimization to maximize the "shufflability" of the instruction stream is to eliminate artificial dependencies, e.g., the Write-After-Write (WAW) and the Write-After-Read (WAR). These dependencies are caused by the limited number of logical registers in the ISAs, which enforces the compiler to reuse the registers during code generation. In the following example, all instructions need to be executed sequentially as there is a true data dependency between the "`LW` and `SUB`" instructions and the "`AND` and `OR`" instructions. Also, the `LW` and `AND` instructions exhibit a WAW dependency, and the `SUB` and `AND` instructions exhibit a WAR dependency. However, the `AND` and `OR` instructions are actually independent from the `LW` and `SUB` instructions thus could be executed before the first two instructions.

```
LW  R1, 4(R1)   // R1 = MEM[R1 + 4]
SUB R3, R1, R2  // R3 = R1 - R2
AND R1, R4, R5  // R1 = R4 & R5
OR  R7, R1, R8  // R7 = R1 | R8
```

We adopt register renaming, a technique widely used in

super-scalar and out-of-order processor design, to address these artificial dependencies (WAW and WAR). The idea is to have more physical registers in the hardware than the number of registers in the ISAs (i.e., logical registers) and reserve new physical registers to be used as destination registers for every instruction that writes to the register file. The current mapping between logical and physical registers is maintained in the renaming unit. The corresponding physical registers for all source and destination registers of all pending instructions are recorded in the corresponding fields in the shuffle buffer. These data are needed to determine unused physical registers, i.e., registers that are not referred to in the logical-to-physical mapping table or in any valid entry in the shuffle buffer.

We use the following example to illustrate the effectiveness of register renaming, where `Xn` denotes physical registers and `Rn` denotes logical registers. Assigning `R1` to `X6` in the `LW` and `SUB` instructions and to `X8` in the `AND` and `OR` instructions yields greater flexibility, regarding the execution order of these instructions. The `AND` instruction can be moved prior to the `LW` or `SUB` instruction, or alternatively, the `SUB` instruction can be moved subsequent to the `AND` or `OR` instruction.

```
LW  X6, 4(X1)   // Initial allocation
SUB X7, X6, X2  // X1=R1 X2=R2 X3=R4
AND X8, X3, X4  // X4=R5 X5=R8
OR  X9, X8, X5
```

Next, we explore other instructions that require special consideration, apart from verifying source/destination registers.

Load and Store[4]**.** The load and store instructions can have an implicit dependency on each other, as the target address is calculated by the value of the register plus an integer offset, which is not known at the instruction fetching stage, until all prior instructions that write to that specific register have been executed. The simplest method is to assume that every load and store depends on all prior loads and stores, which is trivial to implement with low hardware overhead but could incur large run-time overhead. One exception is when there are multiple load instructions in the shuffle buffer without any store instruction. In this case, we can allow these load instructions to be executed in any particular order, given that they don't depend on other instructions in the shuffle buffer.

More challenging, a dependency can exist between "load and store" and "store and store" instructions. For instance, the `SW` instruction in the following code example can't be executed before the first `LW` instruction, as `R1` will contain an incorrect value if `R2+4` equals `R4`. However, the last `LW` instruction can be executed anytime since we can ensure that the `SW` instruction will write to a different memory location (`R4` $\neq$ `R4+4`). Therefore, we can conclude that a dependency exists between a pair of load and/or store instructions when their base register is different, as the register value is not known. When the base register is identical, the offset must be compared depending on whether the instruction worked on a byte (`LB`, `SB`), a half-word (`LH`, `SH`), or a word (`LW`, `SW`).

[4]To support memory-mapped I/O, load and store optimization can be disabled during core configuration or temporarily via the Control and Status Registers (CSR) as described in Sec. V

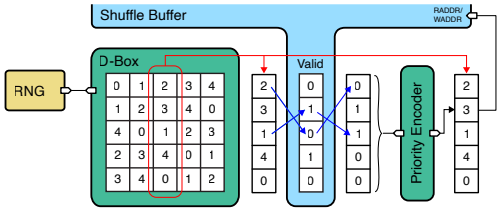

Fig. 5: Instruction selection scheme in ShuffleV.

```
LW  R1, 4(R2)  // R1 = Mem[R2+4]
SW  R3, 0(R4)  // Mem[R4] = R3
LW  R5, 4(R4)  // R5 = Mem[R4+4]
```

Other Instructions. We also need to ensure that synchronization instructions (`FENCE(.I)`), environment instructions (`ECALL` and `EBREAK`) and control status register instructions (`CSR`) are not shuffled around, since they can affect run-time behavior and debugging. To achieve this goal, we set a dependency-bit to cascade these instructions, i.e., these instructions will depend on all prior instructions, and their subsequent instructions will depend on them.

B. Instruction Selection

After filling the instruction buffer, the next step is to pick one ready instruction from the instruction buffer to execute. Although the task appears simple initially, creating random numbers within a certain range and excluding those that correspond to indexes of invalid instructions is not trivial. To solve this problem without introducing extra overhead, ShuffleV employs a rule: it selects the closest ready instruction from the random index. For example, if the random number is 1, we will choose the first valid instruction at index 1, 2 (+1), 0 (-1), 3 (+2), and 4 (-2) in order. Fig. 5 describes how this logic can be implemented in hardware in three steps. (1) We pre-compute the index sort by the distance for each possible random number (0 to $N-1$) and store them in a table called D-Table. (2) We load one column of the D-Table depending on the generated random number and get the valid bit from the corresponding index in the shuffle buffer, as illustrated by the blue arrows in Fig. 5. (3) The priority encoder can be used to find the index of the first '1', which will be used to index a row in the selected D-box column to get the instruction index.

C. Performance Optimization

As a MTD-based defense, the security performance of ShuffleV lies in its shuffling capability, i.e., the randomness of instruction execution. To maximize this, the instruction buffer must be kept full at every cycle that the processor selects an instruction for execution. One exception is in the simplest design without any speculative fetch, where we temporarily allow the instruction buffer to be partially filled when waiting for the control flow instruction (branch or jump) to be executed (see cycle #3-4 in Fig. 6). The drawback is that it requires the core to stall for 0-N cycles to refill the instruction buffer after executing each control flow instruction (see cycle #5-7 in Fig. 6). The number of stall cycles depends on how long the control

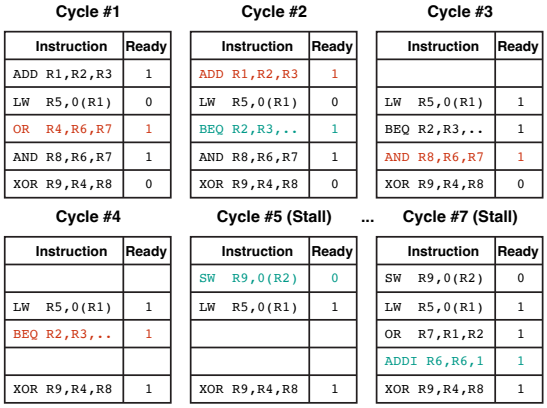

Fig. 6: State of the shuffle buffer during execution without speculative fetch. Instructions highlighted in red are to be removed and executed. Instructions highlighted in green are newly fetched. The ready bit indicates that the entry is valid and is not depend on any other instruction.

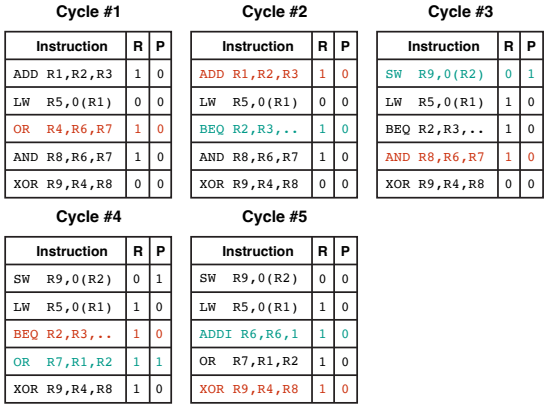

Fig. 7: State of the shuffle buffer during execution with speculative fetch enabled (R="Ready" and P="Prefetch"). Instructions highlighted in red are to be removed and executed. Instructions highlighted in green are newly fetched.

flow instruction stays in the shuffle buffer, as the core won't be able to fetch the next instruction while waiting for this control flow instruction to be executed.

To further improve the performance of ShuffleV, we propose the following three approaches to reduce the stall cycle after each control flow instruction:

① *Speculative fetch*. One effective approach to eliminating stalls after each control flow instruction is to perform speculative fetch, by implementing branch prediction and/or return address stack. A simple implementation is to perform speculative fetch and mark the entry as prefetch, as shown in Fig. 7. When the branch or jump is executed, we can either deassert the prefetch flag to turn the entry into a valid entry and continue execution (see cycle #5 in Fig. 7), or clear all prefetch entries and stall to refill the buffer, depending on whether we predict the target address correctly or not.

② *Speculative execution*. Following the first approach, we can execute all prefetch instructions as if they were valid entries. To achieve this, a checkpoint is created in the same cycle that the branch or jump instruction is added to the shuffle buffer. It stores the current state of the shuffle buffer, the current physical register value, and the current mapping from logical to physical register. If the branch or jump target address is correctly predicted, the checkpoint is discarded and the execution can proceed without any overhead. However, if the target address is incorrectly predicted, we need to restore from the checkpoint and fetch the correct instruction to replace the branch or jump instruction. This approach maximizes instruction sequence randomness in exchange for higher hardware resources for the checkpoint logic.

Before creating the checkpoint, we must ensure that there are no dependent load and store instructions in the shuffle buffer, as the load instruction may retrieve an incorrect value from memory, if the subsequent store instruction was already executed before the rollback to the checkpoint. We illustrate this issue in Fig. 8, where the checkpoint is created in cycle #2, and the `LW` and `SW` instructions are executed in cycles #3 and #4, respectively. Then, in cycle #5, the branch instruction is executed, resulting in a rollback of the core due to a branch mis-prediction. Later in cycle #7, the `LW` instruction executes again and returns an incorrect value, as the succeeding store has already committed to memory.

Also, during execution with an active checkpoint, it is crucial to prevent executing instructions that cannot be rolled back, such as `syscalls` that writes to I/O, etc. This can be achieved by setting the dependency bit of the relevant instructions to the branch or jump instruction. Alternately, we could adopt the load store queue and re-order buffer [42], a hardware unit commonly used in out-of-order processor design, to commit the result of executed instructions in order and enable revert to previous checkpoints. However, doing so restricts the number of CPU cycles an instruction can be shuffled from its initial position. Additionally, committing the results sequentially may increase side-channel leakage.

③ *Modified instruction selection algorithm*. Beside introducing additional hardware for speculative fetch or execution, we can amend the instruction selection module to select the pending branch or jump instruction as soon as it is ready. Therefore, reducing the amount of entry in the instruction buffer that need to be refilled. This method incurs low hardware overhead, but might reduce the randomness of the instruction sequence. Practically, this approach can be combined with the first and second approaches to shorten the speculative fetch and execution duration, thus reducing the rollback penalty when the prediction is incorrect.

D. Additional security feature

To further enhance the execution randomness, ShuffleV also supports inserting dummy instructions by randomly selecting one ALU operation and sending the dummy instruction to the

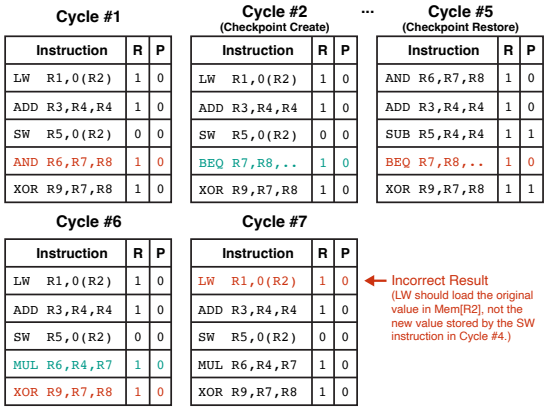

Fig. 8: An example demonstrating the problem with executing dependent load and store instructions during the checkpoint period. `LW` and `SW` are executed in cycles 3 and 4, respectively.

decode/execution stage, for every 0-4, 0-8, or 0-16 instructions. The main difference between ShuffleV and the Secure Ibex [15] is the choice of dummy instructions, ShuffleV uses `ADD`, `AND`, `MUL`, and `MULH` instead of `ADD`, `AND`, `MUL`, and `DIV`. The `DIV` instruction stalls the processor for 37 cycles and is key to the dummy instruction insertion in the Secure Ibex core (which we successfully attacked in Sec. III-C). Since ShuffleV only utilizes dummy as an additional defensive measure, it can avoid using `DIV` instruction to significantly reduce overhead as shown in Fig. 11.

V. ShuffleV: Implementation

A. Design Options

We develop ShuffleV with the following design options, to suit different application scenarios, as well as providing trade-off between security and performance overhead.

- *Optimized memory* (M): this option allows dependencies between load and store to be determined using the logic in Sec. IV-A. Otherwise, the core assumes that all load/store instructions depend on all prior load/store instructions.
- *Decode jump instruction* (J): this option avoids stalling when encountering the `JAL` (jump and link) instruction by adding logic to calculate the jump target immediately.
- *Branch prediction* (B): this option uses branch prediction to reduce stall cycles. We allow only a single branch to be predicted and one checkpoint to be created at a time, to reduce the hardware overhead.
- *Return address stack* (R): this option enables the return address stack to avoid stalling when encountering `JALR` instruction.
- *Multiple checkpoint* (C): this option allows predicting multiple branches and creating multiple checkpoints. It must be specified in combination with the B option.
- *Shortcut branch/jump evaluation* (F): this option forces the instruction selector to select the pending branch or jump instruction as soon as it is ready.

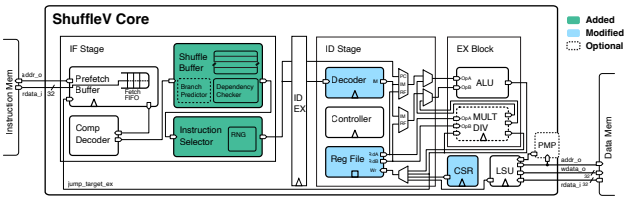

Fig. 9: ShuffleV core block diagram, with variations from the original Ibex highlighted. Green components are newly added. Blue components have been modified. Components with dash border are optional.

B. ShuffleV Simulator

To evaluate the performance and security of ShuffleV, we develop ShuffleV simulator based on libriscv [43], an open-source RISC-V userspace emulator library. ShuffleV simulator supports executing any RISC-V binary on all ShuffleV configurations described in Sec. V-A. It provides detailed execution traces, including internal state of the shuffle logic and reason for each stall, allowing a user to validate the execution randomness and select the most appropriate ShuffleV configuration based on expected workload and performance requirement.

C. Hardware Implementation

To streamline ShuffleV's integration into existing designs, we implemented it on the open-source and popular Ibex RISC-V core [15], an in-order, single-issue core with two pipeline stages. Ibex fully supports the base integer instruction set (RV32I) and can be configured for compressed (RV32C), multiplication and division (RV32M), and bit manipulation (RV32B) extensions. Being interface-compatible with Ibex, ShuffleV can serve as a drop-in replacement in the OpenTitan SoC [16] and several PULP platform SoCs [17].

We chose to extend a simpler in-order core like Ibex over a more complex superscalar out-of-order core for two main reasons. First, this approach allows us to demonstrate the generalizability of our method on a simple, less-expensive core. Second, a more complex architecture would introduce unnecessary components. Despite our modifications, the resulting core is 3.5x smaller than state-of-the-art small out-of-order RISC-V cores, such as [44]. A detailed comparison of the resource utilization can be found in Appendix E.

Fig. 9 highlights the modifications made to the Ibex core. First, the shuffle buffer is introduced alongside the dependency tracking logic, the instruction selection logic, and the random number generator (RNG). The Comp Decoder is moved in front of the shuffle buffer, as our shuffling logic and the ID/EX stage work with instructions in the uncompressed form. Second, we expand the number of registers in the register file to support register renaming and modify the instruction decoder and corresponding data paths in the ID stage to refer to the (renamed) physical registers instead of the logic register specified in the machine code. Third, we add a configuration bit in the Control and Status Registers (CSR) to allow the software developer to enable/disable the

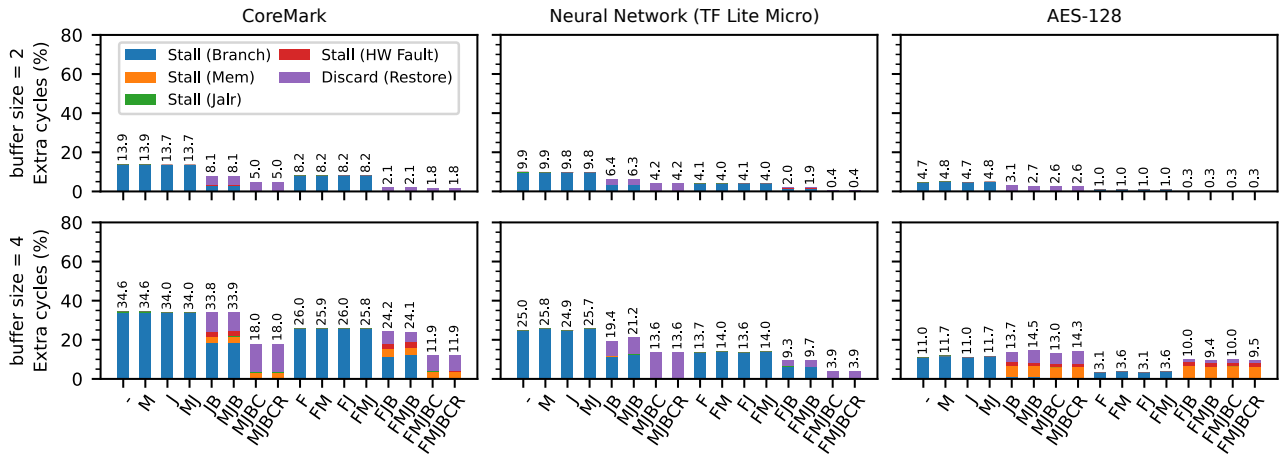

Fig. 10: Performance overhead of different configuration of ShuffleV on the CoreMark benchmark (left), neural network inference on TensorFlow Lite Micro (middle) and AES-128 encryption (right) with shuffle buffer size = 2 and 4

protection to reduce the performance overhead. Finally, we adopt the systemc_rng pseudo-random number generator from OpenCores [45], which combines an LFSR with a CASR based on [46]. Note that the design and evaluation of secure RNGs suitable for FPGA and/or ASIC implementation is out of the scope of this work.

Since our proposed modifications focus on the front-end (i.e., the predict and instruction fetch stages), they are compatible with any single-cycle and pipeline RISC-V core. The performance and security analysis of the implementation on other RISC-V cores are left as future work.

VI. ShuffleV: Evaluation

A. Performance Evaluation

Evaluation Benchmarks. We select three workloads to evaluate ShuffleV, including the standard CPU benchmark (CoreMarks), neural network inference (TF Lite Micro library), and cryptography encryption (AES-128). To demonstrate compatibility with diverse neural network architectures, we employ the ensemble model (Fig. 26 in Appendix D), a small network that consists of all widely used layers and activation function, such as fully connected, convolution, depthwise separable convolution, batch normalization, max pooling, relu, and softmax layers.

Execution Time. We show the overhead of ShuffleV in Fig. 10, which is quantified in percent of extra (stall) cycles to complete the benchmark program on different configurations of ShuffleV with different buffer sizes. These extra cycles result from the shuffle buffer needing to be refilled after certain events. The causes of these stalls are as follows:

- Branch: it occurs when the core pauses fetching while waiting for the branch instruction to be selected for execution (only occurs when branch predictor is not enabled or when multiple checkpoint are not allowed).
- Mem: it occurs when the core pauses fetching while waiting for dependent load and/or store instructions to be selected for execution before the checkpoint can be created (only occurs when branch predictor is enabled).
- Jalr: it occurs when the core pauses fetching while waiting for the `JALR` instruction to be selected for execution.
- HW Fault: this stall is caused by an exception from a speculated instruction. In this case, the core pauses fetching and reverts to the checkpoint.
- Discard: this stall is caused by a misprediction of the branch predictor or the return address stack.

Compared to the baseline unsecured core (Ibex), on the CoreMark benchmark, the execution time overhead ranges from 1.78% to 13.95% and 11.91% to 34.6% when the shuffle buffer size is equal to 2 and 4, respectively. On the neural network workload, the execution time overhead ranges from 0.39% to 9.91% and 3.88% to 25.83% when the shuffle buffer size is 2 and 4, respectively. On the AES encryption workload, the execution time overhead ranges from 0.26% to 4.85% and 3.1% to 14.45% when the shuffle buffer size is equal to 2 and 4, respectively. The overhead when buffer size is equal to 8 is given in Fig. 21 in Appendix B. From our experiment, we found that a buffer size of 4 is a optimal compromise between performance and side-channel security (see Tab. V and VI).

In all workloads, branch instructions are the primary cause of stalls, forcing the core to refill the shuffle buffer as described in Fig. 6. Therefore, enabling branch prediction, such as with the ‘B’ and/or ‘C’ option, is likely to result in lower overhead, as shown in Fig. 10. The ShuffleV simulator and demo SoC currently support four branch prediction algorithms: always taken, always not taken, a static predictor based on branch offset, and a two-bit predictor. From our experiments, static branch prediction performs best for the CoreMark benchmark, achieving 82.00% accuracy. For neural network and AES workloads, the two-bit branch predictor performs best, achieving 85.06% and 71.27% accuracy, respectively. The higher overhead observed in the AES-128 workload when the ‘B’ option is enabled is likely due to low branch prediction

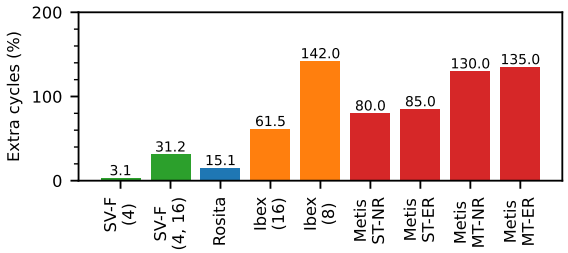

Fig. 11: Execution time overhead on AES-128 encryption on ShuffleV compared with Rosita [9], Secure Ibex [15], and Metis [29] (lower is better). "SV-F (4)"=ShuffleV-F (buffer size=4). "SV-F (4,16)"=ShuffleV-F (buffer size=4, dummy interval=16). "Ibex(16)"=Secure Ibex (dummy interval=16).

accuracy, leading to significant checkpoint restore overhead (purple bar in Fig. 10). Additionally, the high number of dependent memory access instructions in AES-128 prohibits checkpoint creation when branch prediction is enabled (orange bar in Fig. 10). Despite these factors, ShuffleV's overall overhead remains lower than existing work, as demonstrated in Fig. 11. Note that determining the most accurate branch predictor for each specific workload is beyond the scope of this work, and interested readers are advised to refer to [47].

Fig. 11 compares the performance overhead of ShuffleV on AES with SOTA works, including Secure Ibex with dummy instructions [15], Metis [29], the most efficient RISC-V core with code morphing engine, and Rosita [9], an automatic code rewrite engine to protect the masked AES implementation. We choose ShuffleV-F with buffer size equal to 4 for this comparison as it is the configuration used to perform security analysis in Sec. VI-D. When only shuffle is applied, the overhead of ShuffleV is 3.1%, 4.87x less than application specific protection (masked software implementation generated from Rosita) and 21.67x–47.29x less than generalized processors with built-in protection like Secure Ibex and Metis.

When both shuffle and dummy instructions are applied, the overhead of ShuffleV is 2.06x more than Rosita and 2.15x–4.7x less than generalized processors with built-in protection like Secure Ibex and Metis. Therefore, the findings indicate that the overhead of ShuffleV is comparable to the application-specific defense like Rosita [9], while still being universal and user-friendly, eliminating the need for manual adjustments and software modifications.

To further demonstrate the scalability and compatibility of ShuffleV on larger, real-world DNN architectures, Tab. III presents the execution time overhead (in cycles) when executing various models on ShuffleV-F (bs=4) compared to the Ibex core. For all model architectures, we convert predefined models from Keras [48] into TF Lite Micro format and set the input shape to 224x224x3, a common standard for RGB images in recent DNN literature. We include a range of models, from MobileNetV2 with 1.69M parameters (726 million CPU cycles) to ResNet50 with 25.61M parameters (23,045 million CPU cycles). Despite this significant

TABLE III: Execution time overhead of ShuffleV (bs=4) compared to the Ibex core on real-world DNN architecture.

Model Architecture	Model Size		#Extra CPU Cycles
	#Params	#CPU Cycles	
MobileNetV2 (α=0.35)	1.69M	726M	$10.16\%^{\pm 0.0018}$
MobileNetV2 (α=0.75)	2.66M	1,978M	$9.68\%^{\pm 0.0018}$
EfficientNet B0	5.33M	4,175M	$13.15\%^{\pm 0.0010}$
NASNetMobile	5.33M	4,844M	$9.11\%^{\pm 0.0003}$
InceptionV3	23.85M	17,237M	$9.64\%^{\pm 0.0003}$
ResNet50	25.61M	23,045M	$10.02\%^{\pm 0.0004}$

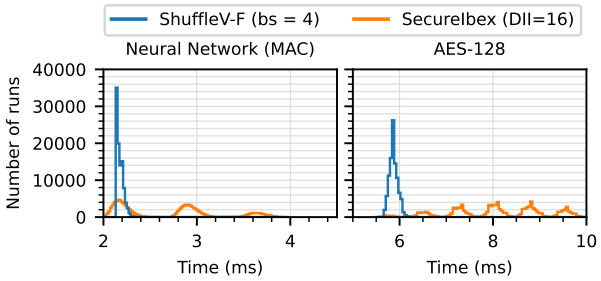

Fig. 12: Distribution of execution time from 100,000 runs on ShuffleV (bs=4) compared with Secure Ibex (DII=16) on MAC and AES-128 workloads.

variation in model size and computational complexity, the results indicate that ShuffleV maintains a consistently low execution time overhead, ranging from 9.11% to 13.15% across all evaluated architectures. This highlights ShuffleV's effectiveness in securing diverse and complex DNN workloads with minimal performance impact.

Unfortunately, it is not possible to compare the performance overhead of neural network workloads, due to the lack of prior related works on countermeasures against EM SCAs on neural networks at the software or microarchitecture level, with the exception of [49], which also did not report performance numbers. We hope that the performance number presented in Fig. 10 and Tab. III could serve as a baseline for other future works on developing countermeasures against EM SCAs on the neural network workload.

Execution Time Variation. While ShuffleV's nondeterministic execution can introduce execution time variation due to shuffle buffer exhaustion and checkpoint restoring, the results in Tab. III demonstrate that this variation is negligible for large program segments. Furthermore, for shorter programs such as AES-128 and 5i5w MAC operations (from Sec. III-C), Fig. 12 clearly illustrates a substantial reduction in execution time variation compared to Secure Ibex. These results demonstrate ShuffleV's suitability for embedded and real-time systems that require predictable run-time behavior. Our instruction shuffling technique offers a distinct advantage over pure dummy instruction insertion, as it dynamically reorders actual program instructions rather than randomly inserting superfluous ones. This approach more efficiently utilizes the processor, leading to reduced execution time overhead and

TABLE IV: FPGA resource utilization of ShuffleV compared to the Ibex [15] and Secure Ibex [15] core. BRAM utilization is 128 in all configurations. ("SV" = "ShuffleV", "bs" = "buffer size" and "di" = "dummy instruction insertion interval")

Configuration	LUT as logic	LUTRAM	FF
Ibex [15]	6547	96	5890
Secure Ibex [15]	6704 (+2.40%)	96	5961 (+1.21%)
SV (bs=2)	7432 (+13.5%)	140	6601 (+12.1%)
SV-MJB (bs=2)	8726 (+33.3%)	140	6896 (+17.1%)
SV-F (bs=2)	7463 (+14.0%)	140	6604 (+12.1%)
SV-FMJB (bs=2)	8599 (+31.3%)	140	6897 (+17.1%)
SV-F (bs=2, di=16)	7598 (+16.1%)	140	6695 (+13.7%)
SV (bs=4)	8032 (+22.7%)	140	6910 (+17.3%)
SV-MJB (bs=4)	8888 (+35.8%)	140	7215 (+22.5%)
SV-F (bs=4)	8079 (+23.4%)	140	6915 (+17.4%)
SV-FMJB (bs=4)	8834 (+34.9%)	140	7218 (+22.6%)
SV-F (bs=4, di=16)	8054 (+23.0%)	140	7011 (+19.0%)
SV (bs=8)	8370 (+27.8%)	140	7552 (+28.2%)
SV-MJB (bs=8)	9829 (+50.1%)	140	7879 (+33.8%)
SV-F (bs=8)	8333 (+27.3%)	140	7560 (+28.4%)
SV-FMJB (bs=8)	9705 (+48.2%)	140	7886 (+33.9%)
SV-F (bs=8, di=16)	8586 (+31.1%)	140	7655 (+30.0%)

improved consistency in execution time.

B. Hardware Resources and Timing

To evaluate ShuffleV hardware resource utilization, power consumption and side channel security, we developed ShuffleV Demo SoC[5] based on the Ibex demo system [40], a simple SoC design that combines the Ibex core with some basic peripheral to support device programming, debugging, and interfacing with external devices. Tab. IV compares resource utilization of notable configurations of ShuffleV Demo SoC against the Ibex Demo SoC. Both SoCs contain the same set of peripherals , differing only in their CPU core. The BRAM utilization is constant across all configurations as it depends solely on the size of the program memory. In all configurations, ShuffleV runs at the same speed as the baseline Ibex core [15] at 50 MHz. It's important to note that the percentage increase in resource utilization is relative to the overall SoC size. For example, Appendix E reveals that the ShuffleV core constitutes only roughly half of the SoC's resources. Therefore, in real-world SoCs with more extensive peripherals and integrated memory, the core will occupy a much smaller proportion of the total area, making its area increase less significant overall.

C. Power Consumption

We assess the effect of our design on power consumption by measuring the average power and energy consumption of a single neural network inference and AES encryption operation. Fig. 13 shows the distribution of the power, energy per inference/encryption, and execution time obtained from 30 runs. In terms of power, both SecureIbex and ShuffleV consume less average power than the unsecure Ibex. In the case of ShuffleV, the reason is due to the additional stall cycles introduced by the shuffling logic, which reduce the

[5]Available at https://github.com/nuntipat/ShuffleV-Demo-System.

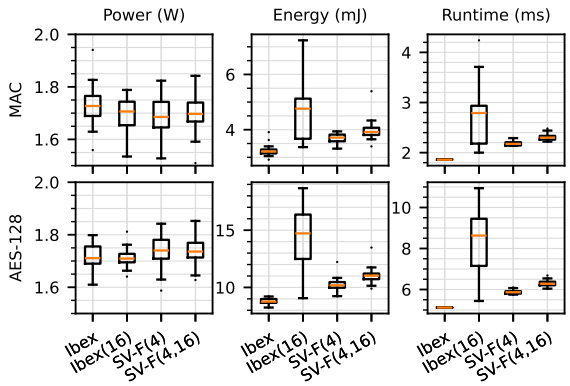

Fig. 13: Power consumption, energy consumption, and execution time of ShuffleV compared with Ibex [15], Secure Ibex (DII=16) (Ibex(16)) [15], ShuffleV-F (bs=4) (SV-F(4)), and ShuffleV-F (bs=4, DII=16) (SV-F(4, 16))

average power. Similarly, Secure Ibex inserts a random number of dummy instructions that take different cycles to execute, resulting in lower average power consumption, e.g., a DIV instruction stalls the fetch and decode stages by up to 37 cycles but increases the energy consumption significantly (see Fig.3 and Fig. 13).

In terms of energy consumption on the neural network workload, the Secure Ibex (Ibex(16)) consumes the highest energy, followed by ShuffleV with dummy (SV-F(4,16)), ShuffleV (SV-F(4)), and the Ibex (4.681mJ vs 3.962mJ vs 3.694mJ vs 3.242mJ). A similar trend can be observed on the AES workload, where the Secure Ibex consumes 14.53mJ, followed by SV-F(4,16) at 11.016mJ, SV-F(4) at 10.242mJ, and Ibex at 8.769mJ. In addition to its lower average energy consumption, ShuffleV achieves a significantly lower variation in energy consumption due to lower execution time variations.

D. Security Evaluation

We test the side-channel resistance of ShuffleV from three perspectives: 1) Overall program shuffle; and 2) Critical operation shuffle; and 3) Resistance against correlation electromagnetic analysis attack (CEMA) using 1M traces.

Overall Program Shuffle. We present the number of ready instructions (in %) in each processor cycle in Fig. 14. The number of ready instructions is defined as the number of instructions that can be selected for execution in each clock cycle, which can range from 1 to N (shuffle buffer's size). The higher number of ready instructions implies higher overall randomness. When executing the neural network workload without applying any optimization (i.e., the vanilla ShuffleV), 28.66% of cycle has 1 ready instruction and 8.45% of cycle has 4 ready instruction, respectively. After enabling the M, J, and B options (Sec. V-A), the percentage of cycles that have with only 1 ready instruction decreases to 7.69%, while the percentage of cycles with 4 ready instructions increases to 28.01%. These experimental results demonstrate the effectiveness of our optimization strategies in Sec. V-A in improving

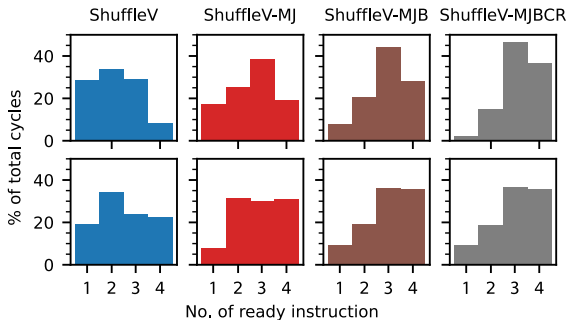

Fig. 14: Proportion of number of ready instructions in each processor cycle while executing the ensemble model on TFLite Micro (top) and AES-128 encryption (bottom) on different configurations of ShuffleV. Results for other configuration are provided in Fig. 22-23 in Appendix C.

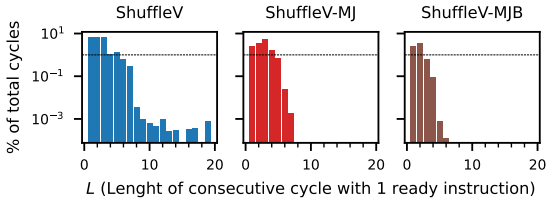

Fig. 15: Percent of total cycles with L consecutive cycles with 1 ready instruction while executing the ensemble model on TFLite Micro. Note that the sum of all bars in each plot is equal to the leftmost bar of the corresponding plot in Fig. 14.

overall randomness. Similar results are observed in the AES workload (Fig. 14 (bottom)).

To further validate the overall randomness, we propose the length of consecutive cycle with 1 ready instruction (L) as another metric to measure how long the core executes instruction without any shuffling. Fig. 15 shows that it is extremely rare for the core to execute without shuffling when performing neural network inference (L>5 cycles for <1% and L>7 cycles for <0.1% of the total cycle). This experiment shows that while ShuffleV may at time be unable to shuffle instructions due to true data dependency (Read-After-Write), a long sequence of non-shuffleable instructions is rare in practice, due to the nature of RISC ISA that requires additional instructions to load/store data from memory, increment the pointer address, etc. These instructions can be reordered to create run-time variation, even though the computation exhibits RAW dependency.

Critical Operation Shuffle. We measure the amount of shuffling for critical instructions, i.e., these directly process secret values. To demonstrates how the overall randomness of the execution weaken the capability of attackers, we define the multiply-accumulate (MAC) operation of the neural network inference, as well as the load/store operation between the S-box and the state array as the critical operation. Using the shuffle buffer size equal to 4 and 8, Fig. 16 illustrates that

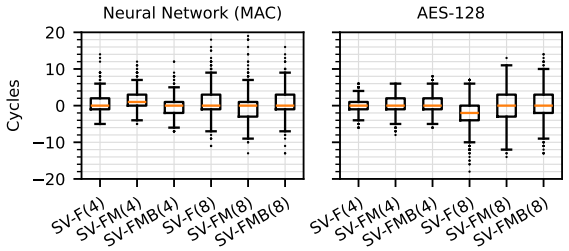

Fig. 16: The amount of shuffling (in cycles) of the critical instructions in neural network inference (left) and AES-128 encryption (right) on different configurations of ShuffleV.

the amount of shift for critical operations is approximately -13 to +19 cycles on the neural network workload, and -18 to +14 cycles on the AES-128 workload, depend on the ShuffleV configuration and the overall program duration. These results demonstrate that, although ShuffleV targets randomizing the overall program, it also affects the critical operations and can help adverse the ability of attack to conduct a successful attack.

Resistance against CEMA. We validate the performance of ShuffleV by conducting CEMA attack on the multiply-accumulate (MAC) operation and the SubBytes operation in the first round of AES-128 encryption. The measurements are conducted on the PYNQ-Z2 FPGA board running the ShuffleV Demo SoC at 50MHz. We follow the experimental setup in Sec. III-B. Tab. V shows the CEMA results on the AES-128 encryption. From our experiment, the vanilla ShuffleV (SV-F4) with 3.1% performance overhead can protect approximately 12 out of 16 key bytes (Appendix. A Fig. 17). The number of traces required to perform the attack and the index of a byte in the key that can be successfully attacked change between subsequent runs, which proves that ShuffleV execution is random. Note that even though several key bytes may leak in each run, it is still difficult for attackers to combine results from multiple runs to get the correct key as they do not know which byte is corrected. When considering ShuffleV with dummy instructions, the performance overhead is 31.2% and all except the first byte of the key can be successfully protected (Appendix. A Fig. 19). This is due to the limited shuffling space between the first byte and the trigger signal. Note that in this experiment, we assume a very strong attacker who can place the trigger signal precisely at the beginning of the encryption process, which is not feasible in practice. Thus, the actual attack success rate will be much lower.

We show the CEMA results on the MAC in Tab. VI. When considering ShuffleV with a buffer size of 4 (i.e., SV-F (4)), the performance overhead is 13.7%, and all weights from the fifth weight onwards can be successfully protected. The first weight can only be attacked in conjunction with an attack on weights 1 and 2. Attacking weights 2-4 yields five candidates with equal correlation values, resulting in a 20 percent success rate (Appendix. A Fig. 18). Again, the reason that weights 1–4 can be attacked is due to their close proximity to the trigger signal, and the success rate in reality will be lower.

TABLE V: Results of performing a CEMA against AES-128 on unprotected Ibex [15], Secure Ibex [15], ShuffleV-F with buffer size = 4 (SV-F (4)) and ShuffleV-F with buffer size = 4 and dummy interval = 16 (SV-F (4,16)). The percentage after the configuration name indicates the overhead compared to the Ibex core. ✗ indicates an unsuccessful attack at 1M traces.

Config \ Byte#	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
Ibex	300	300	400	150	400	200	100	100	150	250	250	250	1050	100	450	450
Secure Ibex (61.5%)	80k	51k	93k	31k	53k	43k	18k	10k	10k	13k	13k	10k	12k	3k	14k	1k
SV-F (4) (3.1%) (1st run)	235k	✗	385k	✗	✗	✗	✗	✗	✗	✗	✗	✗	170k	340k	310k	✗
SV-F (4) (3.1%) (2nd run)	✗	✗	✗	✗	✗	✗	✗	✗	710k	✗	✗	✗	360k	✗	100k	530k
SV-F (4,16) (31.2%)	660k	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

TABLE VI: Results of performing a CEMA against MAC operation on unprotected Ibex [15], Secure Ibex [15], ShuffleV-F (buffer size = 4) and ShuffleV-F (buffer size = 4, dummy interval = 16). The percentage after the configuration name indicates the overhead compared to the Ibex core. ✗ indicates an unsuccessful attack at 1M traces. *1The attack was performed on weights 1-2 simultaneously. *2The attack yields five candidates with equal correlation (20% success rate).

Config \ Weight#	1	2	3	4	5
Ibex	270	170	300	150	670
Secure Ibex (37.5%)	37k*1	37k	10k	14k	36k
SV-F (4) (13.7%)	130k*1,2	130k*2	45k*2	100k*2	✗
SV-F (4,16) (41.8%)	✗	✗	✗	✗	✗

For better defense, we test ShuffleV with dummy instruction enabled (i.e., SV-F (4, 16)), which has a performance overhead of 41.8% and could protect all weights (Appendix. A Fig. 20). Given that real-world neural networks have much more than 5 weights, the results show that both configurations can successfully protect neural network execution, and the choice of which configuration to use depends on the requirement and acceptable performance loss.

VII. Discussion and Future Work

In this section, we discuss the limitations and potential misconceptions of ShuffleV, proposing mitigation strategies and directions for future work.

Compiler optimization: A modern compiler may reorder instructions to maximize performance for target microarchitectures, e.g., the load instruction may be placed apart from the instructions that require the loaded value to prevent pipeline stall. In some cases, ShuffleV may unintentionally void this optimization, resulting in performance loss. Further studies could be conducted to enhance the compatibility between ShuffleV and compiler optimizations.

Deterministic execution for embedded system: Many embedded and real-time systems require predictable run-time behavior, i.e., low variation between runs. ShuffleV improve upon SecureIbex significantly in this regard, as shown in Fig. 12 and 13 (right) by avoiding using long-running instructions, e.g., DIV as a dummy, and by adopting several strategies to prevent shuffle buffer stalls (see Sec. IV-C). If more control is needed, ShuffleV allows developers to disable the protection through the Control and Status Registers when handling interrupts or performing time-critical operations (see Sec. V-C).

ShuffleV vs other side-channel attacks: Although our threat model focuses on single-tenant embedded systems, which are not susceptible to software-based attacks like Spectre, we anticipate that ShuffleV's non-deterministic execution and memory access pattern could extend its security benefits to mitigate timing and cache-based side-channel attacks. Applying ShuffleV to desktop-class processors to validate this potential is a key direction for future research.

ShuffleV vs Out-of-Order core: ShuffleV and traditional out-of-order and super-scalar cores are fundamentally different, since ShuffleV reorders instructions randomly, whereas traditional out-of-order cores reorder instructions deterministically, i.e., they shuffle instructions in the same way across multiple runs, making them still vulnerable to CEMA attacks.

VIII. Conclusion

This paper introduces ShuffleV, a microarchitectural defense strategy against EM side-channel attacks (SCAs) in microprocessors. Employing the moving target defense (MTD) philosophy, ShuffleV mitigates EM SCAs by randomly shuffling program instruction execution order and inserting dummy instructions. Unlike application-specific countermeasures, ShuffleV offers automatic protection without algorithm modification or software recompilation. To accommodate diverse design needs, ShuffleV provides various configurations enabling trade-offs between performance overhead and security. We developed an ShuffleV simulator for rapid evaluation of different configurations, allowing users to analyze execution traces of any RISC-V binary. We implemented ShuffleV on the open-source Ibex RISC-V core, enabling its use as a drop-in replacement in existing SoC designs. Furthermore, we developed an ShuffleV Demo SoC and implemented it on a Xilinx XUP PYNQ-Z2 FPGA board to assess hardware resource footprint, performance, power consumption, and EM SCA resistance. Experimental results demonstrate ShuffleV's successful protection of neural network model confidentiality and AES encryption keys with low overhead in performance, energy, and hardware resources.

Acknowledgements

This work is supported in part by the U.S. National Science Foundation under Grants CNS-2153690, CNS-2247892, CNS-2239672, OAC-2319962, and CNS-2326597.

References

[1] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer, 1999, pp. 388–397.

[2] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, ser. Lecture Notes in Computer Science, vol. 3156. Springer, 2004, pp. 16–29. [Online]. Available: https://iacr.org/archive/ches2004/31560016/31560016.pdf

[3] S. Ors, F. Gurkaynak, E. Oswald, and B. Preneel, "Power-analysis attack on an asic aes implementation," in *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, vol. 2, 2004, pp. 546–552 Vol.2.

[4] C. Luo, Y. Fei, P. Luo, S. Mukherjee, and D. Kaeli, "Side-channel power analysis of a gpu aes implementation," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, 2015, pp. 281–288.

[5] L. Batina, S. Bhasin, D. Jap, and S. Picek, "Csi nn: Reverse engineering of neural network architectures through electromagnetic side channel," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 515–532.

[6] G. Takatoi, T. Sugawara, K. Sakiyama, and Y. Li, "Simple electromagnetic analysis against activation functions of deep neural networks," in *Applied Cryptography and Network Security Workshops*. Cham: Springer International Publishing, 2020, pp. 181–197.

[7] N. Dalvi, P. Domingos, Mausam, S. Sanghai, and D. Verma, "Adversarial classification," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 99–108. [Online]. Available: https://doi.org/10.1145/1014052.1014066

[8] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson *et al.*, "Extracting training data from large language models," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2633–2650.

[9] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: Towards automatic elimination of power-analysis leakage in ciphers," in *Proceedings 2021 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, 2021.

[10] T. B. Singha, R. P. Palathinkal, and S. R. Ahamed, "Securing aes designs against power analysis attacks: A survey," *IEEE Internet of Things Journal*, pp. 1–1, 2023.

[11] A. Dubey, R. Cammarota, and A. Aysu, "Maskednet: The first hardware inference engine aiming power side-channel protection," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 197–208.

[12] ——, "Bomanet: boolean masking of an entire neural network," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3400302.3415649

[13] Y. Liu, D. Dachman-Soled, and A. Srivastava, "Mitigating reverse engineering attacks on deep neural networks," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 657–662.

[14] Y. Luo, S. Duan, C. Gongye, Y. Fei, and X. Xu, "Nnrearch: A tensor program scheduling framework against neural network architecture reverse engineering," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–9.

[15] "Ibex risc-v core," https://github.com/lowRISC/ibex, 2015.

[16] "Opentitan-open source silicon root of trust (rot) project," https://github.com/lowRISC/opentitan, 2019.

[17] "Pulp platform," https://www.pulp-platform.org/, 2013.

[18] EEMBC, "CoreMark: An eembc benchmark," 2024. [Online]. Available: https://www.eembc.org/coremark/

[19] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, "Advanced encryption standard (aes)," 2001-11-26 2001.

[20] Google LLC, "TensorFlow Lite Micro," 2024. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers

[21] L. Gerlach, D. Weber, R. Zhang, and M. Schwarz, "A security risc: Microarchitectural attacks on hardware risc-v cpus," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2321–2338.

[22] J. Danial, D. Das, S. Ghosh, A. Raychowdhury, and S. Sen, "Scniffer: Low-cost, automated, efficient electromagnetic side-channel sniffing," *IEEE Access*, vol. 8, pp. 173 414–173 427, 2020.

[23] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, "A more efficient aes threshold implementation," in *Progress in Cryptology – AFRICACRYPT 2014*, D. Pointcheval and D. Vergnaud, Eds. Cham: Springer International Publishing, 2014, pp. 267–284.

[24] ——, "Trade-offs for threshold implementations illustrated on aes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 7, pp. 1188–1200, 2015.

[25] T. Cnudde, B. Bilgin, O. Reparaz, V. Nikov, and S. Nikova, "Higher-order threshold implementation of the aes s-box," in *Revised Selected Papers of the 14th International Conference on Smart Card Research and Advanced Applications - Volume 9514*, ser. CARDIS 2015. Berlin, Heidelberg: Springer-Verlag, 2015, p. 259–272.

[26] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *Information and Communications Security*, P. Ning, S. Qing, and N. Li, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 529–545.

[27] H. Gross, S. Mangard, and T. Korak, "Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order," in *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security*, ser. TIS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 3. [Online]. Available: https://doi.org/10.1145/2996366.2996426

[28] A. G. Bayrak, N. Velickovic, P. Ienne, and W. Burleson, "An architecture-independent instruction shuffler to protect against side-channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, jan 2012.

[29] F. Antognazza, A. Barenghi, and G. Pelosi, "Metis: An integrated morphing engine cpu to protect against side channel attacks," *IEEE Access*, vol. 9, pp. 69 210–69 225, 2021.

[30] H. Gross, M. Jelinek, S. Mangard, T. Unterluggauer, and M. Werner, "Concealing secrets in embedded processors designs," in *Smart Card Research and Advanced Applications*, K. Lemke-Rust and M. Tunstall, Eds. Cham: Springer International Publishing, 2017, pp. 89–104.

[31] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2: Lightweight authenticated encryption and hashing," *J. Cryptol.*, vol. 34, no. 3, p. 33, 2021. [Online]. Available: https://doi.org/10.1007/s00145-021-09398-9

[32] "Freertos," https://www.freertos.org/, 2003.

[33] "Zephyr," https://zephyrproject.org, 2014.

[34] M. M. Ahmadi, F. Khalid, and M. Shafique, "Side-channel attacks on risc-v processors: Current progress, challenges, and opportunities," *arXiv preprint arXiv:2106.08877*, 2021.

[35] A. Gonzalez, B. Korpan, J. Zhao, E. Younis, and K. Asanovic, "Replicating and mitigating spectre attacks on an open source risc-v microarchitecture," in *Third Workshop on Computer Architecture Research with RISC-V (CARRV 2019)*, 2019.

[36] "Xup pynq-z2," https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html, 2017.

[37] "4 series mso mixed signal oscilloscope," https://www.tek.com/en/products/oscilloscopes/4-series-mso.

[38] "Rf-b 0.3-3," https://www.langer-emv.de/en/download/35/855/17/rf-b-0-3-3-h-field-probe-mini-30-mhz-up-to-3-ghz.pdf.

[39] "Pa 303 sma set," https://www.langer-emv.de/en/product/preamplifier/37/pa-303-sma-set-preamplifier-100-khz-up-to-3-ghz/520.

[40] "Ibex demo system," https://github.com/lowRISC/ibex-demo-system.

[41] A. Waterman and K. Asanovi, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, 2019.

[42] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ser. ISCA '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 291–299. [Online]. Available: https://doi.org/10.1145/285930.285988

[43] A.-A. Walla, "libriscv," https://github.com/fwsGonzo/libriscv, 2023.

[44] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, K. Inoue, and R. Shioya, "An open source fpga-optimized out-of-order risc-v soft processor," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 63–71.

[45] J. Castillo Villar, "Systemc/verilog random number generator," https://opencores.org/projects/systemc_rng, 2004.

[46] T. E. Tkacik, "A hardware random number generator," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '02. Berlin, Heidelberg: Springer-Verlag, 2002, p. 450–453.
[47] S. Mittal, "A survey of techniques for dynamic branch prediction," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 1, p. e4666, 2019.
[48] F. Chollet *et al.*, "Keras," https://keras.io, 2015.
[49] M. Brosch, M. Probst, and G. Sigl, "Counteract side-channel analysis of neural networks by shuffling," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 1305–1310.
[50] C. Celio, D. A. Patterson, and K. Asanović, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," Tech. Rep. UCB/EECS-2015-167, Jun 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html
[51] R. Höller, D. Haselberger, D. Ballek, P. Rössler, M. Krapfenbauer, and M. Linauer, "Open-source risc-v processor ip cores for fpgas — overview and evaluation," in *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, 2019, pp. 1–6.

APPENDIX A
CEMA ATTACK RESULTS

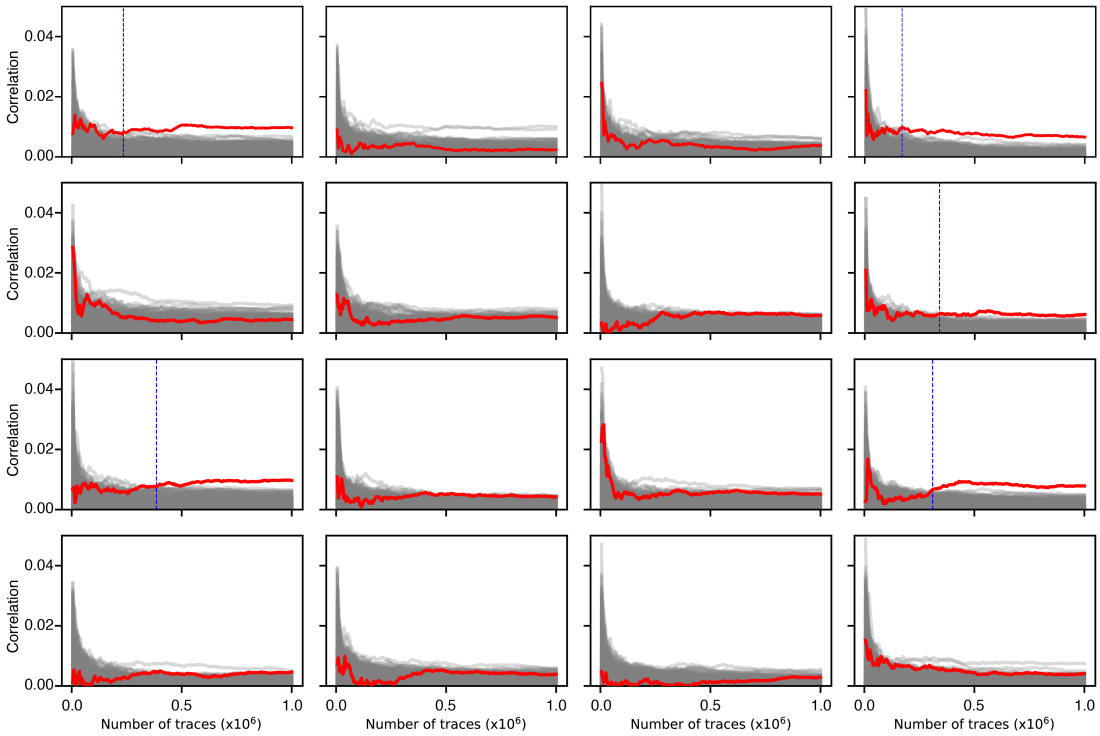

Fig. 17: Result of performing a CEMA against AES on the ShuffleV-F core (buffer size = 4). Blue dashed lines indicate the number of traces required for the successful attack. Otherwise, the attack is unsuccessful at 1M traces.

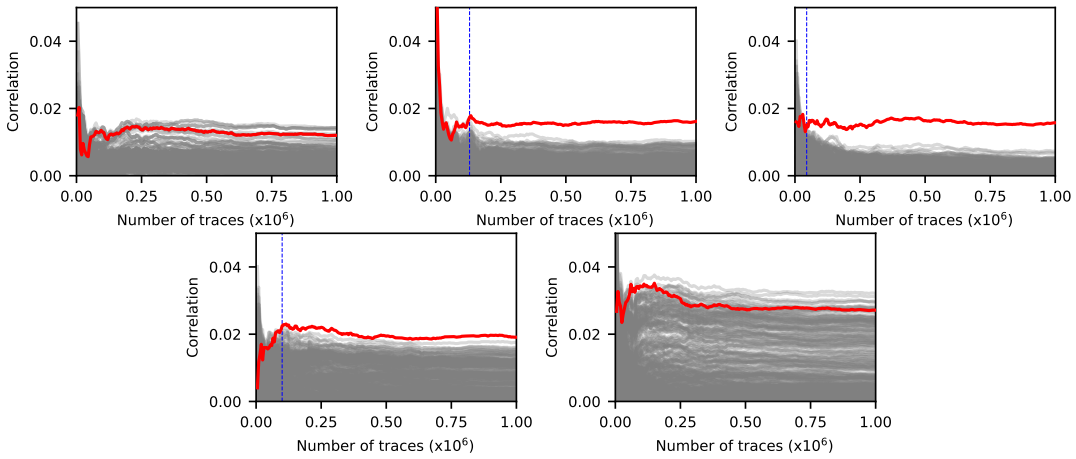

Fig. 18: Result of performing a CEMA against 5i5w MAC on the ShuffleV-F core (buffer size = 4). Blue dashed lines indicate the number of traces required for the successful attack. Otherwise, the attack is unsuccessful at 1M traces.

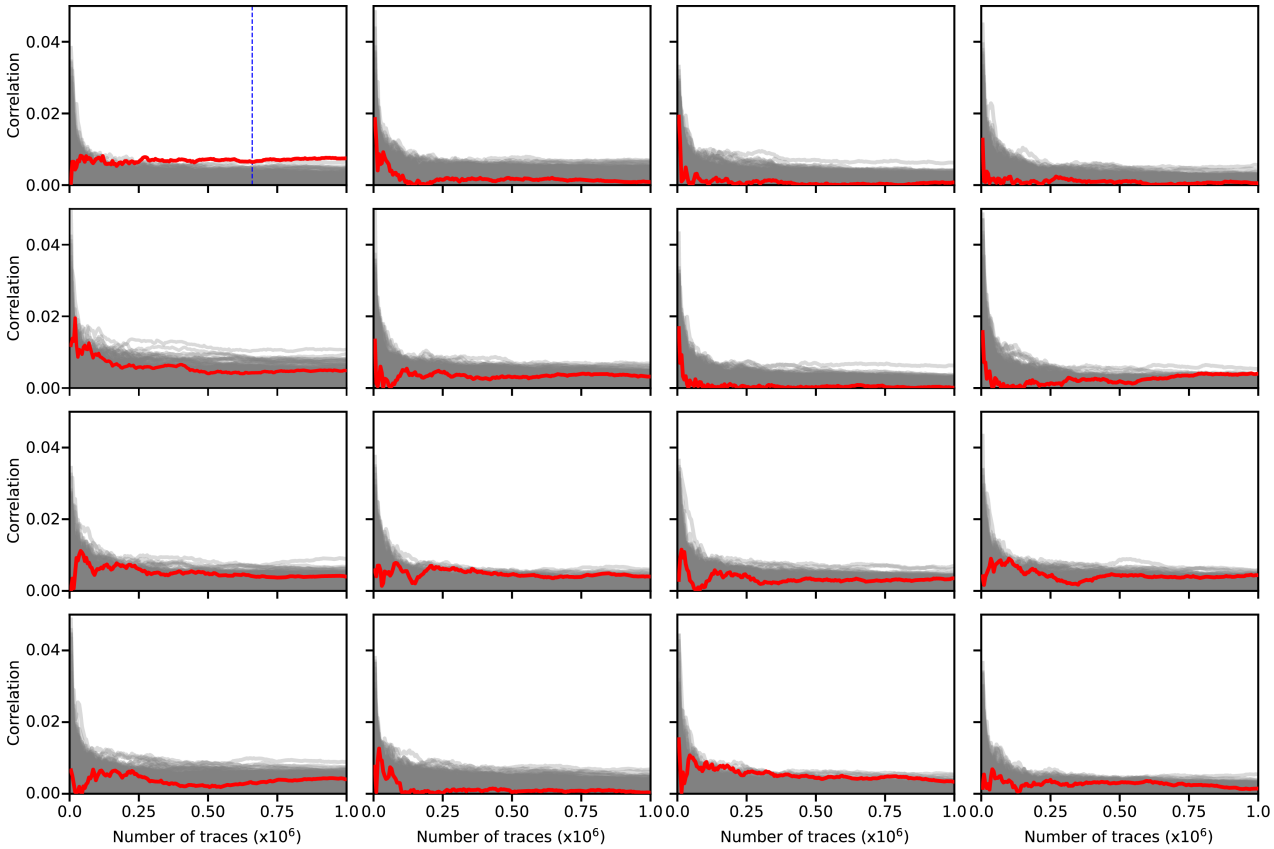

Fig. 19: Result of performing a CEMA against AES on the ShuffleV-F core (buffer size = 4, dummy interval = 16). Blue dashed lines indicate the number of traces required for the successful attack. Otherwise, the attack is unsuccessful at 1M traces.

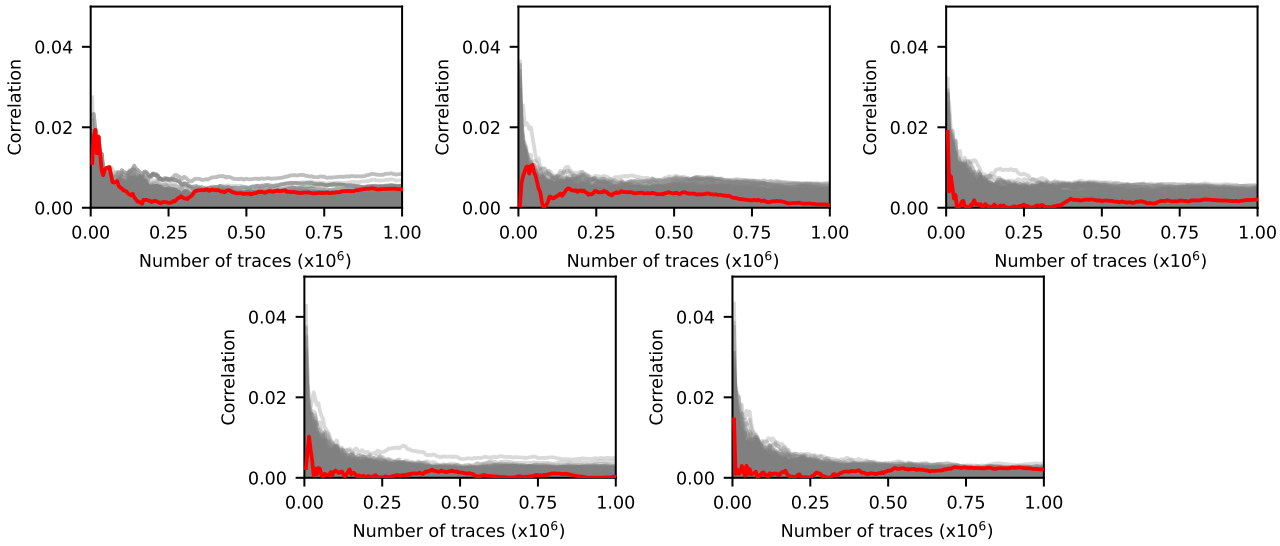

Fig. 20: Result of performing a CEMA against 5i5w MAC on the ShuffleV-F core (buffer size = 4, dummy interval = 16).

APPENDIX B
PERFORMANCE OVERHEAD

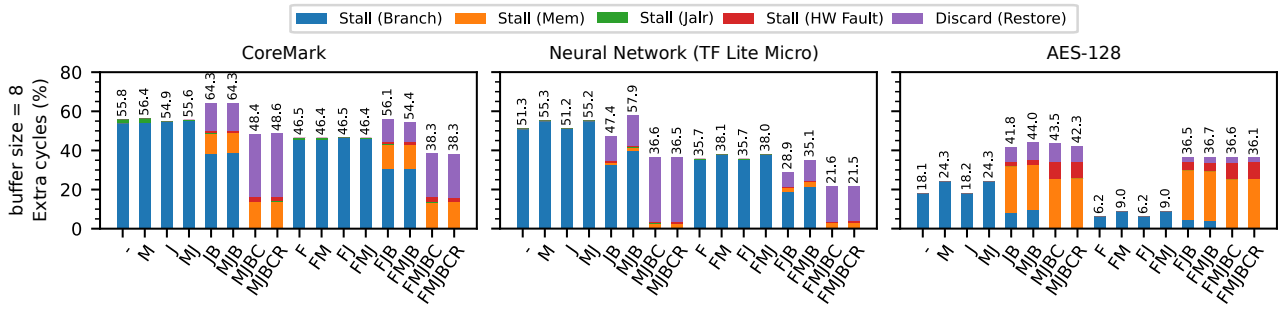

Fig. 21: Performance overhead of different configuration of ShuffleV on the CoreMark benchmark (left), neural network inference on TensorFlow Lite Micro (middle) and AES-128 encryption (right) with shuffle buffer size = 8

APPENDIX C
SHUFFABLITY METRICES FOR ALL CONFIGURATIONS

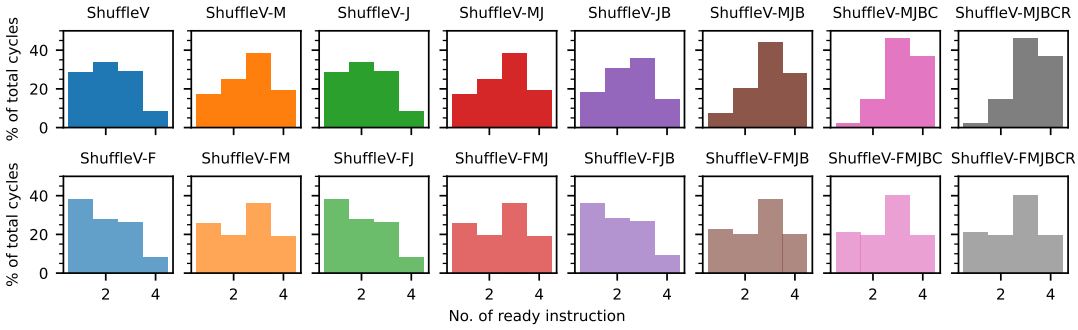

Fig. 22: Proportion of CPU cycle with 1 - 4 ready instruction while executing the ensemble model on TFLite Micro

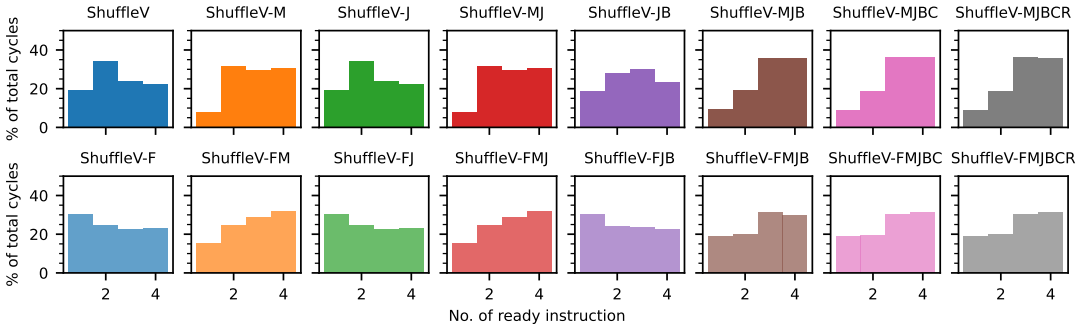

Fig. 23: Proportion of CPU cycle with 1 - 4 ready instruction while executing the AES-128 encryption

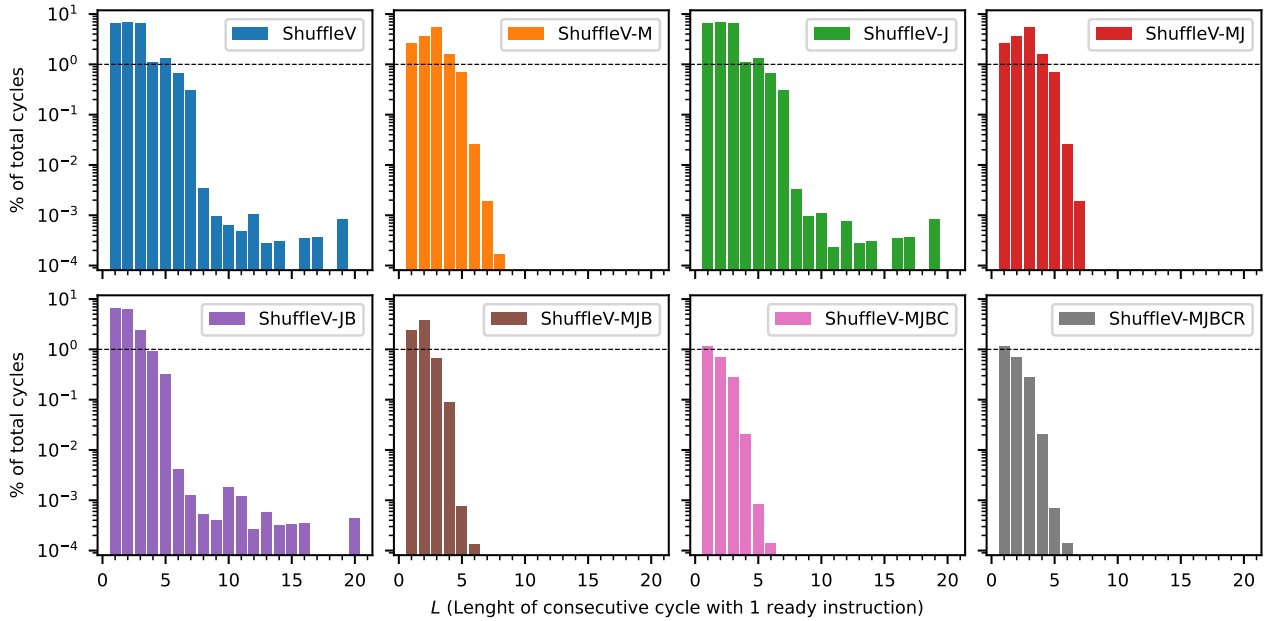

Fig. 24: Percent of total cycles with L consecutive cycles with 1 ready instruction while executing the ensemble model on TFLite Micro. Note that the sum of all bars in each plot is equal to the leftmost bar of the corresponding plot in Fig. 22.

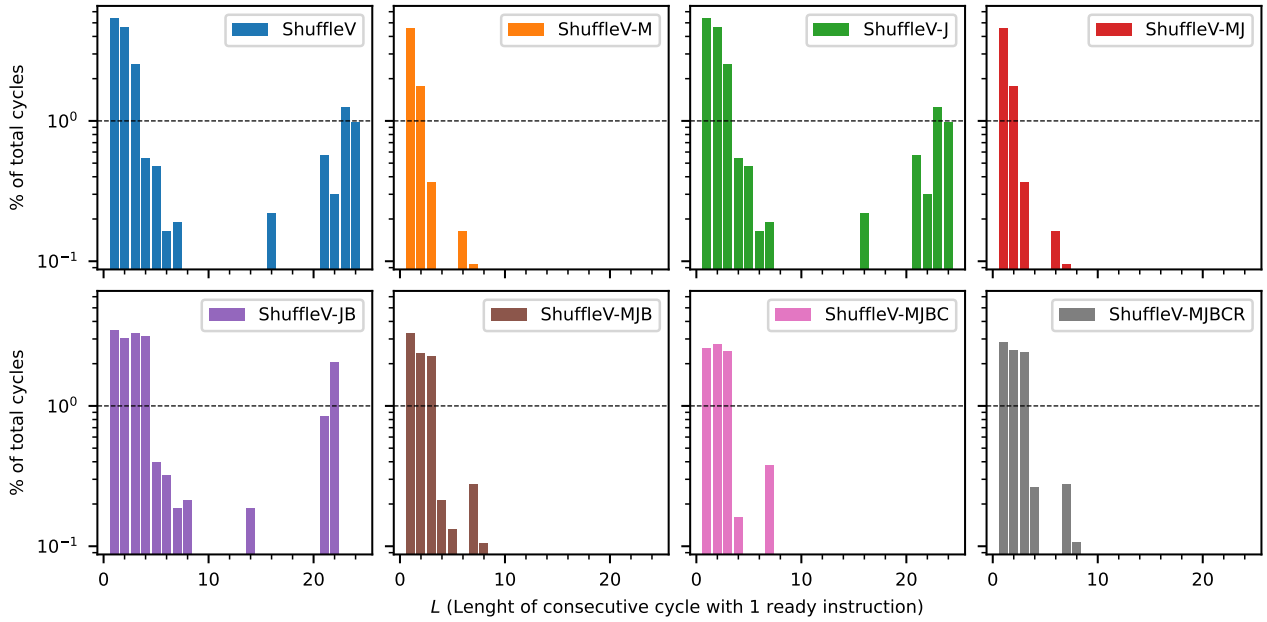

Fig. 25: Percent of total cycles with L consecutive cycles with 1 ready instruction while executing the AES-128 encryption. Note that the sum of all bars in each plot is equal to the leftmost bar of the corresponding plot in Fig. 23.

APPENDIX D
ENSEMBLE MODEL ARCHITECTURE

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 5)         140

batch_normalization (BatchNormalization)  (None, 30, 30, 5)  20

separable_conv2d (SeparableConv2D)  (None, 28, 28, 5)  75

average_pooling2d (AveragePooling2D)  (None, 14, 14, 5)  0

separable_conv2d_1 (SeparableConv2D)  (None, 10, 10, 1)  131

max_pooling2d (MaxPooling2D)  (None, 5, 5, 1)          0

flatten (Flatten)            (None, 25)                0

dense (Dense)                (None, 5)                 130

=================================================================
Total params: 496
Trainable params: 486
```

Fig. 26: Architecture of the ensemble model used in our performance analysis.

APPENDIX E
FPGA RESOURCE UTILIZATION

TABLE VII: FPGA resource utilization of ShuffleV compared to other RISC-V cores. [*1]The Xilinx Zynq XC7020 FPGA combines an Arm Cortex-A9 processor with Artix-7 based programmable logic thus resource utilization from both FPGA devices should be comparable. [*2]The BOOM result is assumed to be based on BOOMv1 [50]. [*3]The OPA and BOOM results are approximated from the figure presented in [44].

Core	ISA	Type	FPGA Device[*1]	LUTs	Registers	Reference
PicoRV32	RV32IM	In-order	Xilinx Artix XC7A35T	1,765	1,075	[51]
Ibex	RV32IMC	In-order	Xilinx Zynq XC7Z020	3,161	1,933	-
ShuffleV (bs=4)	RV32IMC	Out-of-order (Nondeterministic)	Xilinx Zynq XC7Z020	4,411	2,993	-
RI5CY	RV32IMC	In-order	Xilinx Artix XC7A35T	6,748	2,577	[51]
RSD	RV32IM	Out-of-order	Xilinx Zynq XC7Z020	15,379	8,584	[44]
OPA[*3]	RV32IM	Out-of-order	Xilinx Zynq XC7Z020	20,500	9,800	[44]
BOOM[*2*3]	RV32IMAC	Out-of-order	Xilinx Zynq XC7Z020	43,600	21,500	[44]