VeCT: Secure and Efficient Constant-Time Code Rewriting with Vector Extensions

Qisheng Jiang
Duke University
qisheng.jiang@duke.edu

Danfeng Zhang
Duke University
danfeng.zhang@duke.edu

Abstract

Timing channels allow attackers to extract secrets by analyzing the execution time of a victim program. Constant-time (CT) disciplines enforce security against timing attacks via data-flow/control-flow linearization (DFL/CFL). However, the rewritten constant-time code typically considerably increases the memory footprint of the original code, causing significant overhead. We present VeCT, a compiler-based code rewriter that leverages vector extensions to retain constant-time guarantees while improving performance. We first apply rigorous statistical tests to derive practical "safe-use" rules for AVX-512 instructions whose implementation details are proprietary; this analysis also reveals a previously unknown vulnerability in a state-of-the-art constant-time rewriter. Guided by these rules, VeCT introduces a novel strategy that eliminates unnecessary data loads in rewritten code, and enables vectorization to further improve efficiency. We implement VeCT based on LLVM to automatically transform code into AVX-512-based constant-time equivalents. On real-world applications like AES and Blowfish, VeCT reduces the overhead of transformed code by up to 98.9% compared to the state-of-the-art, while preserving constant-time behavior.

1 Introduction

Timing attacks can rapidly extract confidential information by analyzing the execution time of a security-critical system. For instance, many modern cryptographic implementations are vulnerable to timing attacks [3, 5, 18, 27]. To mitigate timing channels, one common practice is to identify and rule out *dangerous code patterns* that lead to timing channels. Notably in cryptographic systems, a common countermeasure against timing attacks is to follow *constant-time (CT) disciplines* [1, 6], which rule out (1) branching on secret-dependent data, as well as (2) accessing memory with secret-dependent offsets (e.g., an array access with a secret-dependent index).

In order to eliminate CT violations without changing program semantics, a commonly adopted strategy is *Linearization*, which may be implemented either through manual intervention, or by means of automatic code rewriting tools [4, 22, 34, 35, 39]:

- Data-Flow Linearization (DFL) eliminates memory accesses with secret-dependent offset. For example, a secret-dependent memory access `x = A[s]` where `s` is secret can be transformed to

```
for (i=0; i<size; i++) {x = (i==s)? A[i]:x;}
```

  where `size` is the length of array `A`. Note that the transformed code has the same semantics as the original code, but it enforces constant-time discipline as it touches all elements in array `A` regardless of the value of `s`, where the conditional assignment can be realized as a constant-time instruction such as `cmov` on x86 [7].

- Control-Flow Linearization (CFL) eliminates branching on secret-dependent data. For example, a secret-dependent branch

```
if s==0 then x=A[1] else y=A[10]
```

  may be transformed to

```
x = (s==0)? A[1]:x;
y = (s!=0)? A[10]:y;
```

  Similar to the DFL example above, the transformed code is CT as it executes the same sequence of CT instructions regardless of the secret value of `s`.

While prior works present several automatic linearization techniques to rewrite code to obey constant-time disciplines [4, 35, 39], they all incur significant performance overhead compared with the original code. For example, we observe 3.6× to 70.5× overhead in our evaluation (Section 6). The reason behind the significant overhead is unsurprising when we take a closer look at the secure code after linearization: both DFL and CFL considerably increase the memory footprint of the original code; they touch *all memory*

addresses that could be accessed with different secret values. Such a set includes the whole array `A` in the DFL example and {`A[1]`, `A[10]`} in the CFL example above. Prior work formalized the set as a *differential set* and showed that differential set size is usually large in complex and realistic code [22].

In this paper, we explore a novel approach to boost the performance of constant-time code, by utilizing Vector Extensions (VE). Vector Extensions are a set of Single Instruction Multiple Data (SIMD) operations supported by mainstream modern CPU architectures. By design, VE instructions significantly increase data throughput and reduce the number of instructions needed for tasks such as vector arithmetic, matrix operations, and signal processing. Hence, VE technology is widely adopted in scientific computing, machine learning, and multimedia applications, with support in modern compilers and optimized libraries, making it a key enabler of high-performance software.

While VE can enable high-performance software, this paper presents the first *secure and efficient* application of VE in the security domain: demonstrating that VE instructions can enable both secure and efficient mitigation of timing channels. We focus on Intel's Advanced Vector Extensions 512 (AVX-512), a representative VE with wide vector registers and rich instruction sets, offering superior vector processing capabilities. The key insights that we explore in this paper are that (1) VE instructions *hide* some side-channel information compared with their load/store counterparts, and (2) secure code after linearization can be optimized with VE support. To build on the insights, we overcome several key challenges:

- Since the implementation details of AVX-512 are not publicly documented, the timing and cache behaviors of AVX-512 instructions were previously unknown. To address this challenge, we conduct a reverse-engineering analysis of AVX-512 instructions and utilize principled statistical testing [17, 31] to uncover potential side channels inherent in AVX-512 implementations. Guided by the outcomes of this study, we uncover a set of security rules for utilizing AVX-512 for side-channel mitigation, and also identify a previously unknown security vulnerability in a state-of-the-art constant-time code rewriter [4].
- After linearization, a naive AVX-512-based implementation incurs substantial overhead. For example, a CT implementation for `A[s] = 10` where `s` is secret requires three steps: (1) loading all existing values (as vectors) of array `A`, (2) among the vectors, modifying the bits corresponding to `A[s]` to value `10`, and (3) writing the vectors back to array `A`. We present novel code rewriting strategies (such as modifying the memory layout to eliminate step (1) above, and packing consecutive memory accesses into one vector instruction) to improve performance while maintaining CT properties.
- Compiler assistance is required to detect, after linearization, which instructions can be vectorized to meet both security and performance goals. To tackle this challenge, we built VeCT on top of LLVM [21] and Constantine [4], enabling automatic transformation of the original code into an AVX-512-based CT equivalent that adheres to AVX-512 security guidelines while also exploiting the performance optimizations identified in this paper.

We evaluate the performance of VeCT on both microbenchmarks and real-world applications. Compared to a state-of-the-art CT rewriter, VeCT showed significant overhead reductions. Across all real-world benchmarks, our gather/scatter-based approach reduced overhead by an average of 46.0% (up to 98.8%), while our packed load/store-based approach achieved an average reduction of 42.9% (up to 98.9%).

2 Background and Motivation

2.1 Timing Side Channels

Timing side-channel attacks have emerged as particularly pernicious threats, posing serious risks to system security [15]. Timing side channels arise when attackers exploit secret-dependent variations in program execution [42], including (1) the overall execution time of a program, and (2) observable memory access behavior, such as cache usage, pipeline behavior, and cache bank contention [23, 41]. These leaks allow attackers to infer sensitive information by observing external timing or memory access patterns.

The following example code illustrates a side channel caused by a secret-dependent branch in an old OpenSSL implementation of sliding window exponentiation [42]. The result of `BN_is_bit_set` decides whether to execute the following multiplication function `BN_mul`, leading to an execution time difference between the two sides of the branch.

```
1 if (BN_is_bit_set(p, i))
2     if (!BN_mul(rr, rr, v, ctx)) goto err;
```

The next example code illustrates another side channel due to secret-dependent memory accesses in a BearSSL implementation of AES [28]. Accessing an S-box element in memory fetches a corresponding cache line, creating a side channel that an attacker can exploit to recover the secret `s0`.

```
1 v0 = SboxExt0( s0 >> 24 ) ^ ...
```

2.2 Vector Extensions

Mainstream modern CPU architectures incorporate vector extensions, such as Intel AVX [13], ARM SVE [2], and RISC-V "V" [32], to support single instruction multiple data (SIMD) operations, enabling data parallelism and improved performance. In this paper, we select Intel's Advanced Vector Extensions (AVX) family as a representative of VE. Specifically, we target AVX-512 for its superior vector processing capabilities over earlier AVX variants. Figure 1 depicts the major

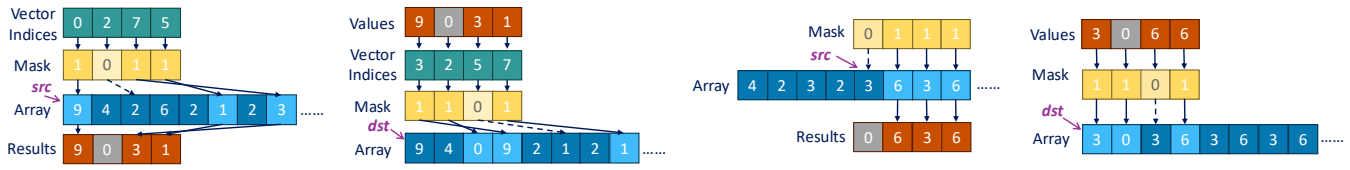


Figure 1: An Illustration of Vector Memory Instructions. (a) Gather loads elements from non-contiguous memory locations using `indices` based on `src`, with a `mask` determining which values are kept in `results`; (b) Scatter stores elements to memory based on `dst` in a reverse way of gather; (c) Packed Load reads a block of contiguous memory from `src` into `results` with a `mask` selection; and (d) Packed Store writes `values` into contiguous memory from `dst` with a `mask` selection. In all cases, the array resides in memory, while all other operands are held in vector or general-purpose registers.

memory operations in AVX-512 that we utilize in this paper. The *Vector Mask Register* enables selective memory load/store on specific elements. The operation is performed on the corresponding element only if the corresponding mask bit is set to one. Otherwise, the operation is suppressed. **Packed Load and Store** operations can access packed and contiguous data elements in a single instruction, while **Gather and Scatter** may access elements in non-contiguous memory locations based on specified *vector indices*. Taking the scatter operation in Figure 1b as an example, the first vector index 3 specifies that the first element 9 in source register is stored to the position 3 in the array.

2.3 Constant-Time Code Rewriting

Constant-time (CT) programming [4, 42] is a collection of programming principles to guide programmers to thwart timing attacks. Specifically, to comply with CT programming, programs should always have (1) no secret-dependent branch (*control flow*), and (2) no secret-dependent memory access (*data flow*), regardless of their secret inputs. However, it is error-prone and daunting for programmers to write CT code manually [4]. Thus, CT code rewriting tools have been developed to automatically transform programs into their CT equivalents. Existing code rewriters [4, 35, 39] utilize Control Flow Linearization (CFL) and Data Flow Linearization (DFL) to eliminate sensitive branches and memory access patterns with the following interfaces:

```
ct_select(taken, t, f);
ct_load  (src, set, set_size);
ct_store (dst, value, set, set_size);
```

CFL removes CT-violating branches by introducing a `taken` predicate that tracks the real value of the branch condition, executing both sides of the branch, and merging the execution results with `ct_select`. The `ct_select`, a constant-time selection instruction, returns `t` if the `taken` is `true`; otherwise, it returns `f`. DFL replaces each secret-dependent memory access with `ct_load/store` in the original code, ensuring all possible addresses of such a sensitive memory access (i.e., *differential set* [22]) are touched after linearization. A common implementation of `ct_load` and `ct_store` accesses each and every element in the address set `set`. Since such possible addresses are often contiguous [23], `set` and `set_size` represent the start address and its range. For instance, Listing 2 shows a linearized equivalence of the original code in Listing 1, where `c` is a secret and can be revealed through a timing channel. DFL guarantees all elements in the differential set (e.g., `in`) are touched for each load so that attackers cannot distinguish which element is accessed.

Naively, DFL can be achieved with a simple for-loop that iterates over every element in `set`. A `ct_load` sequentially accesses each candidate address, selecting the value from the target and ignoring the values for all others. For a `ct_store`, it first loads all possible elements into registers (i.e., *Preemptive Data Load*), and then only the target value is updated with the new data while keeping others unchanged. Finally, all values are written back uniformly. More efficient linearization techniques are also introduced in prior work [4, 35].

2.4 Vector Extension-Based DFL

To alleviate the performance overhead of the naive for-loop-based method, Constantine [4] incorporates AVX vectorization with a striding strategy, enabling parallel memory accesses and reducing the number of load/store instructions. It supports two common variants: (1) Gather/Scatter version and (2) Packed Load/Store version.

However, Constantine's naive adoption of AVX vectorization has two notable limitations. First, a *security* concern: **the constant-time guarantees of AVX memory instructions remain insufficiently studied**. Masked AVX memory instructions may vary the set of accessed addresses based on mask values, leading to potential side channels. In fact, we demonstrate in Section 4.4 that Constantine's generated CT code is still vulnerable to timing attacks, which defeats the purpose of CT code rewriting. Without a comprehensive analysis, their impact on constant-time behavior cannot be conclusively determined, and they may potentially introduce side channels. Second, an *efficiency* concern: **the capability**

```
if (c < 28) {
  out = in[c];
} else {
  out = in[c - 28];
}
```

Listing 1: Original Code

```
taken = c < 28;
pt  = ct_select(taken, &in[c], NULL);
bt  = ct_load(pt, in, sizeof(in));
pf  = ct_select(!taken, &in[c - 28], NULL);
bf  = ct_load(pf, in, sizeof(in));
out = ct_select(taken, bt, bf);
```

Listing 2: Rewrited Code with DFL-then-CFL

```
taken = c < 28;
pt  = ct_select(taken, &in[c], NULL);
pf  = ct_select(!taken, &in[c - 28], NULL);
[bt, bf] = vct_load([pt, pf], in, sizeof(in
    ));
out = ct_select(taken, bt, bf);
```

Listing 3: Rewrited Code with CFL-then-DFL

of AVX to fully optimize DFL remains underexplored. For instance, masked store instructions can selectively update elements within a vector, yet Constantine does not fully exploit this feature, resulting in redundant memory operations and traffic. Moreover, conventional single-element CT memory access interfaces hamper the utilization of parallelism available in AVX-512 instructions, resulting in suboptimal performance. These limitations motivate the need for a redesigned AVX-based DFL that preserves constant-time guarantees while eliminating redundant memory accesses.

3 Threat Model

As a constant-time rewriter, the transformed code must adhere to established CT security principles, including:

- The program's control flow must not depend on secret values to prevent attacks by measuring *execution time* or *instruction-cache access patterns*.
- The memory addresses accessed during execution must not depend on secret data to defend against attacks based on *memory access patterns*.
- The program should choose data operand independent timing (DOIT) instructions [11], the *execution time* of which does not depend on the values of their operands.

Following these principles and prior work [1, 4, 42], we assume a strong adversary model, which is capable of executing arbitrary code concurrently with the victim program, even including on the same physical or logical core. Additionally, the adversary has access to the source code or binary of the target program and tries to extract sensitive information by exploiting microarchitectural side channels.

In particular, we assume the attacker can perform three kinds of timing attacks in the literature. The first: ($\mathfrak{T}_1$) **Execution Timing-Based Attacks** measure the execution time of the entire *victim program* or specific instructions of it.

Instead of measuring the timing of the victim program, the other two kinds of timing attacks measure indirect effects of secret-dependent memory accesses of the victim program:

$\mathfrak{T}_2$ **Cache-Level Attacks**: These attacks leverage spatial leakage at the granularity of cache lines (e.g., 64 bytes). The adversary observes whether or not a specific cache line was accessed, such as Flush+Reload [40], Prime+Scope [29], and Prime+Probe [26].

$\mathfrak{T}_3$ **Word-Level Attacks**: Side channels can emerge at word granularity (e.g., 4 bytes), for example through cache bank conflicts [41] or false dependencies between loads and stores [24]. While cache bank conflicts are architecture-dependent and have not been observed on recent processors [24], we consider the potential threat and adopt MemJam [24] to test if AVX instructions are vulnerable to word-level attacks in this paper.

4 CT Guarantees for AVX-512 Memory Access

Although AVX-512 memory instructions provide highly efficient data parallelism with SIMD, their susceptibility to side-channel attacks (e.g., whether the mask value of an AVX-512 instruction can be revealed via a timing channel) has not yet been systematically investigated in the literature.

In this section, we perform principled statistical tests on a series of microbenchmark experiments to unearth whether mask registers and vector indices of AVX-512 instructions can affect execution time, cache-level state, and word-level state. The main contributions of the study are: (1) it provides a reusable statistical testing framework for understanding CT guarantees for vector extension implementations on specific hardware, (2) it discovers the following CT guarantees of Intel AVX-512 memory instructions,[1] which can serve as the security guidelines for using them in CT implementations, and (3) it reveals that a naive adoption of AVX-512 in prior work [4] is indeed *vulnerable* to timing attacks.

Takeaway

$\mathfrak{G}_1$ **Packed Load** instruction guarantees CT properties even when mask bits are secret-dependent.
$\mathfrak{G}_2$ **Packed Store** instruction guarantees CT properties as long as at least one bit is set in the mask.
$\mathfrak{G}_3$ **Gather / Scatter** instruction with all-one mask guarantees CT properties as long as each cache line is uniformly accessed regardless of the index's value.

[1] We focus our study on Intel processors, given their prominence among architectures supporting vector extensions. Nevertheless, our t-test framework is broadly applicable to other architectures. A comprehensive evaluation across additional platforms is deferred to future work.

4.1 Methodology

To test whether each mask and vector index, when applicable, can be revealed via a timing channel, we adopt a statistical hypothesis test that tries to disprove the null hypothesis that "the two timing distributions are equal" given two different masks, or indices. Specifically, we follow `dudect` [31] and use an online Welch's t-test [17] to assess the null hypothesis stated above. Recall that a timing attack manifests in three forms (Section 3). The t-test checks against each of them:

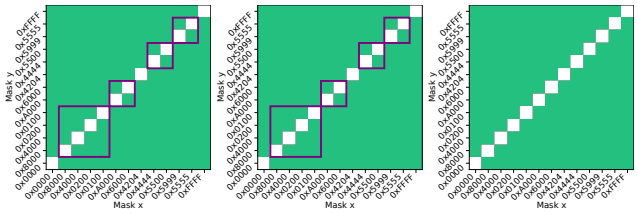
- Execution time ($\mathfrak{T}_1$): Measuring the execution time of the target AVX-512 instruction under varying setups.
- Cache flush time ($\mathfrak{T}_2$): Measuring the flush time for each potentially accessed cache line after executing the target AVX-512 instruction to determine which memory regions were touched during execution.
- Word-level access time ($\mathfrak{T}_3$): Measuring the access latency of potentially conflicting memory addresses after executing the target AVX-512 instruction with various setups. In particular, we follow the state-of-the-art attack MemJam [24] to test if word-level microarchitectural conflicts (e.g., false dependencies or cache bank contention) exist. As this threat model is stronger than other models above, we evaluate $\mathfrak{T}_3$ only for those setups that remain CT under $\mathfrak{T}_1$ and $\mathfrak{T}_2$.

Each instruction was evaluated 100,000 times for every configuration. An elevated t-statistic t from the t-test indicates a potential CT violation. Following [31], we adopt a threshold of 10: $t \leq 10$ suggests constant-time, while $t > 10$ indicates potential violation. We further distinguish samples with $t > 500$ solely to highlight areas with high statistical confidence. For readability, we normalize the t-statistic to 0, 1, and 2 according to the three categories above, which are represented in heatmaps as **Green**, **Orange**, and **Red**, respectively. We perform the t-test on two different Intel CPU platforms, including Intel Core i9-11900 and Xeon Gold 5215, and the CT guarantees above hold for both. Due to space constraints, we present representative results on i9-11900. The complete set of results is included in Appendix A.

4.2 Mask Register

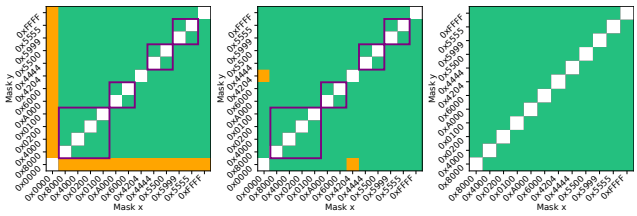
To evaluate how the mask register affects CT properties of AVX-512 masked memory instructions, we design a test suite based on two primary dimensions of mask values: (1) The number of bits set to one (i.e., how many elements are accessed), and (2) The positions of the one-bits in the mask (i.e., which set of elements are accessed). These two dimensions respectively capture both *quantitative* and *structural* variations in the mask.

Recall that each packed load/store instruction processes sixteen 32-bit integer elements within a single cache line



(a) Execution. (b) Cache Flush. (c) Word Access.

Figure 2: T-test Results for Packed Load in a Single Cache Line under Varying Masks. Inside the Purple Boxes: masks with the same number of 1-bits, but at different positions.



(a) Execution. (b) Cache Flush. (c) Word Access.

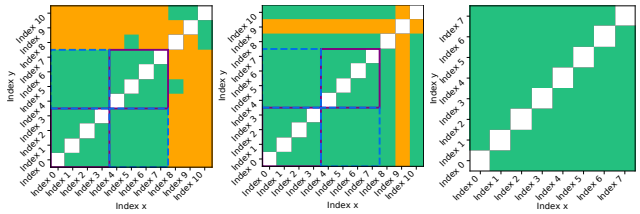
Figure 3: T-test Results for Packed Store in a Single Cache Line under Varying Masks. Inside the Purple Boxes: masks with the same number of 1-bits, but at different positions.

by design (Section 2.2). Each gather/scatter instruction is more complicated: it also operates on sixteen vector indices that might span multiple cache lines. For the *single-cache line setup*, we confine all indices to the range 0 to 15. For the *multiple-cache line setup*, we use indices $[0, 1, 16, 17, 32, 33, 48, 49, 64, 65, 80, 81, 96, 97, 112, 113]$ to spread accessed elements across 8 consecutive cache lines.

The t-test results are visualized in the heatmaps that we explain next, where each cell represents the t-statistic between a pair of masks. To aid interpretation, we highlight two specific classes using **Purple Boxes**: (1) *Cells within boxes* correspond to mask pairs with the same number of enabled bits (1-bit count) but different positions. This allows us to isolate the effect of position-dependent behavior. (2) *Cells outside boxes* correspond to mask pairs with different 1-bit counts, where both the number and position of accesses vary.

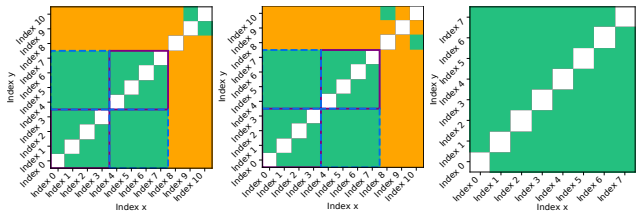
Packed Load. The t-test shows minimal variation under all three threat models: execution time, cache flush time, and word-level access time. As visualized in Figure 2, all t-statistics are below 10 (the actual t-values are all well below 10) regardless of mask value. This indicates that neither the number of accessed elements nor their positions impact observable timing behavior. Thus, we conclude that masked packed loads are constant-time.

Packed Store. In contrast, packed stores do exhibit mask-dependent execution time and cache behavior. As shown in Figures 3a and 3b, the all-zero mask (i.e., `0x0000`) clearly exhibits a distinct timing behavior when compared with other



(a) Execution. (b) Cache Flush.[2] (c) Word Access.

Figure 4: T-test Results for Gather in Multiple Cache Lines under Varying Vector Indices. Purple Boxes and Blue Dashed Boxes both isolate index groups for the same set of cache lines. The index groups within Purple Boxes in addition have an identical number of accesses per cache line.



(a) Execution. (b) Cache Flush.[2] (c) Word Access.

Figure 5: T-test Results for Scatter in Multiple Cache Lines under Varying Vector Indices. Purple Boxes and Blue Dashed Boxes both isolate index groups for the same set of cache lines. The index groups within Purple Boxes in addition have an identical number of accesses per cache line.

masks. Our hypothesis is that the Intel processor implements hidden optimizations for all-zero masks, causing distinct timing and cache behavior. That said, all *non-zero* masks do exhibit CT properties under all threat models.

Gather and Scatter. Unlike packed load/store, gather/scatter operations show stronger correlation between their timing behaviors and mask values under the t-test: both quantitative and structural variations in mask values violate CT properties in both single- and multiple-cache line setups (results are included in Appendix A). These effects, especially when spanning multiple cache lines, may be observable via execution-time or cache-flush measurements. Our hypothesis is that the implementations of masked gather and scatter instructions *selectively* access elements based on the mask, causing the timing dependency on masks. *To securely use gather/scatter operations in CT implementations, we should set masks to be all-ones* (i.e., `0xFFFF`).

4.3 Vector Indices

To investigate the CT properties of vector indices used in **gather and scatter** operations, we conducted experiments with all-one masks (i.e., `0xFFFF`) while varying the vector indices. For the t-test, we select eleven representative index groups, each consisting of sixteen indices, which are carefully designed to explore two types of variation: (1) Inter-cache line *quantitative* variation (i.e., varying how many elements are accessed in each line), and (2) Intra-cache line *structural* variation (i.e., accessing different elements in the same cache line). Due to space constraints, all index groups used for testing are shown Table 2 in Appendix A.

The t-test results are visualized in Figures 4 and 5 for both gather and scatter instructions. Similar to the results we showed earlier, here **Purple Boxes** highlight index group pairs where only the positions of accessed elements within a cache line are changed. In addition, **Blue Dashed Boxes** indicate pairs where the number of accessed elements per cache line is changed, while they access the same set of cache lines.

We first observe that position-only changes (see purple boxes in Figures 4 and 5) lead to no timing difference, indicating that accessing different elements in the same cache line *preserves* CT properties. Moreover, the same applies to the number of accessed elements per cache line, as long as each cache line is accessed at least once. This is indicated by the low t-values within the blue dashed boxes. In addition, repeated accesses to the same location (e.g., Index 3 and 7 issue repeated accesses to locations $[0, 32, 64, 96]$ twice and three times, respectively) do not violate CT properties, as long as each cache line is accessed at least once.

On the other hand, we observe that CT properties are violated among index groups that access different sets of cache lines, as indicated by the orange cells outside the boxes. In conclusion, gather/scatter operations are CT with respect to index variations, as long as every cache line is touched.

4.4 Case Study: CT Violation in Constantine

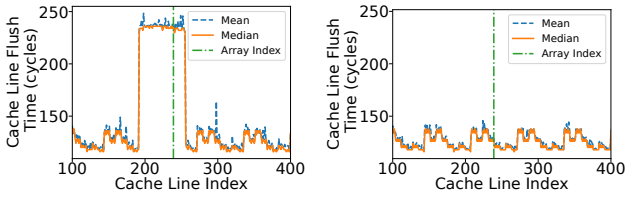
An important consequence of the investigation into CT guarantees is the identification of a CT violation arising from the use of AVX-512 instructions in Constantine [4], a state-of-the-art CT code rewriter. For example, the code below shows both the original non-CT version (left) where a secret-dependent condition `c` determines whether to store `v` to the `i`-th element of the array `out` or not. The corresponding transformed code with Constantine is shown to the right.

```
// original code
if (c > 0)
  out[i] = v;
```

```
taken = c > 0; // transformed code
p = ct_select(taken, &out[i], NULL);
ct_store(p, v, out, sizeof(out));
```

Constantine's implementation of `ct_store` directly modifies memory using masked scatter instructions. This design requires a precise mask to avoid corrupting memory, where only the single bit for the target address is enabled. If `taken` is false, the address `p` resolves to `NULL`, which results in all-zero masks due to no address match. Otherwise, the corresponding bit in the mask is set to one. This transformation clearly

[2]We present the results for line 7, and the rest are shown in Appendix A.



(a) Access with Index `i`. (b) Access with `NULL` Pointer.

Figure 6: The Flush Time of Each Cache Line in the Data Flow Linearized Array by Constantine.

violates CT guarantee $\mathfrak{G}_3$ (Section 4.2). In fact, our evaluation shows that the execution time when `taken` is `true` is approximately 10.3× of that when `taken` is `false`.

To confirm CT violation on cache effects, we respectively measured the flush times of each cache line associated with array `out` for both cases: `taken` = `true` and `taken` = `false`. The results are shown in Figure 6. There is an evident difference in flush times by comparing these two cases, revealing the secret value of `c`. Moreover, the peak for `taken` = `true` also reveals the range of the accessed index `i`. Consequently, the transformed code is vulnerable to timing attacks.

5 Design of VeCT

In this section, we design VeCT, a secure constant-time code rewriter with high efficiency using vector extensions. Figure 7 shows the workflow of VeCT, which rewrites an unsafe program to its CT equivalent. Initially, VeCT performs *constant-time violation detection* via information flow analysis [4, 42] (①, see Section 5.1) to detect program statements that violate CT properties (②). Then it applies Control-Flow Linearization (③, see Section 5.1) and adopts *vectorization identification* to identify vectorizable regions (④, see Section 5.4.1) in the code. Next, VeCT linearizes data flow (⑤) by replacing vulnerable memory accesses with CT equivalents, including both single-element accesses (⑥, see Section 5.3) and vectorized accesses (⑦, see Section 5.4) identified in the previous stage. Finally, VeCT *generates* the constant-time version of executable code with LLVM (⑧).

5.1 CT Violation Detection and Linearization

Extensive research has been dedicated to detecting CT violations (e.g., [4, 36, 42]), in terms of both secret-dependent branches and memory accesses, and automatically eliminating them by CFL and DFL (e.g., [4, 22, 35, 39]). Since VeCT builds upon established methods for violation detection and CFL, we do not elaborate on these aspects here and instead, direct interested readers to the corresponding prior work.

We do note that VeCT deliberately performs CFL before DFL in order to optimize performance without compromising security. To illustrate why, consider a code snippet from the DES algorithm in LibTomCrypt [19], as shown in Listing 1, where both sides of a branch access the same array but with different indices. A *DFL-then-CFL* approach would handle each branch separately, resulting in two distinct CT memory access instructions after the branch is resolved (see Listing 2). In contrast, our *CFL-then-DFL* approach first linearizes the control flow. This allows us to merge the memory accesses from both branches into a single, vectorized CT memory instruction, as shown in Listing 3. This transformation, without altering program semantics, reduces the number of memory operations from two to one. The result is a considerable reduction in total memory traffic and an improvement in execution efficiency, all while fully preserving the CT guarantees.

Next, we elaborate our contributions on DFL.

5.2 Baseline: a Naive Adoption of AVX-512

Constantine [4] is the first work that utilizes AVX-512 in CT rewriting. However, as shown in Section 4.4, its naive adoption of AVX-512 leads to CT violations, which defeats the purpose of CT code rewriting. To fix this vulnerability, we first present a naive solution based on the existing rewriting strategies in Constantine:

- To implement `ct_load(src,set,set_size)`, use either Packed Load or Gather operation with all-one masks and vector indices covering all cache lines (for Gather). Then use vector operations to extract the value of `src`.
- To implement `ct_store(dst,value,set,set_size)`, first use Packed Load or Gather to load all elements as vectors, similar to `ct_load` above. Then, in the vectors, update the corresponding value at the target address `dst` to `value`. Finally, use either Packed Store or Scatter operation with all-one masks and vector indices covering all cache lines (for Scatter) to store the updated vectors back to memory.

Compared with Constantine [4], the baseline here only modifies the implementation of scatter-based `ct_store` with an extra load step, which we term *preemptive data loads*. This is the same strategy used by the packed store-based `ct_store` in Constantine. As the changes are minor, we refer to this baseline as Constantine+ hereafter.

5.3 VeCT Rewriting Strategy

We first detail CT interfaces `ct_load` and `ct_store` in VeCT. Informed by the CT guarantees developed in this paper (Section 4), we observe the following opportunities for performance improvement:

- Constantine+'s implementation of `ct_store` employs *preemptive data loads* prior to stores to thwart the timing attacks demonstrated in Section 4.4. However, doing so

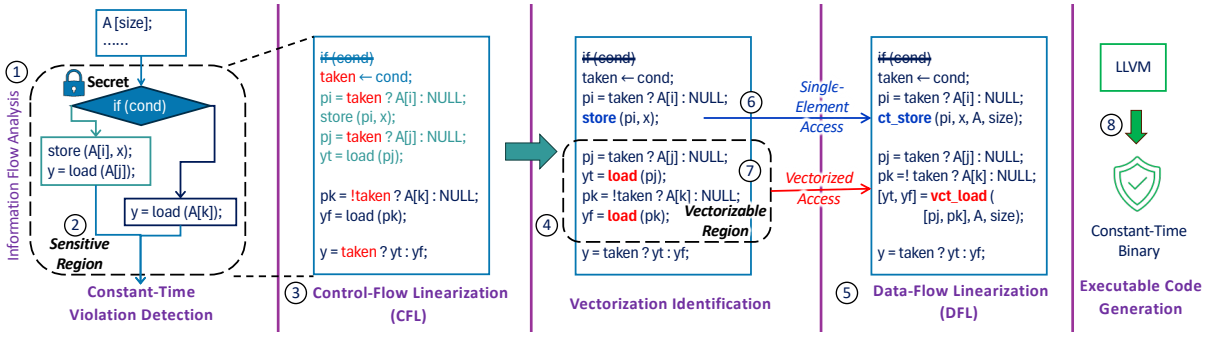


Figure 7: An End-to-End Rewriting Procedure for VeCT.

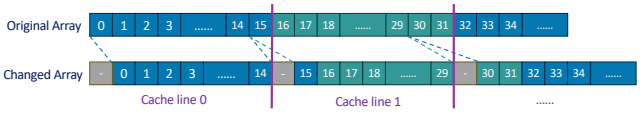


Figure 8: Changed Memory Layout in VeCT.

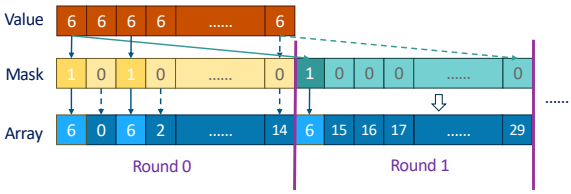


Figure 9: Packed Store-based CT Store in VeCT.

also incurs considerable overhead. Based on CT guarantees $\mathfrak{G}_2$ and $\mathfrak{G}_3$, we observe that the preemptive data loads can be safely eliminated as long as at least one bit is set in the mask of Packed Store, or as long as each cache line is uniformly accessed for Scatter with an all-one mask. Accordingly, VeCT modifies the memory layout so that at least one dummy element per cache line is being written to.

- Constantine+'s implementation of `ct_load` conservatively loads all elements with Packed Load. Based on CT guarantees $\mathfrak{G}_1$, we observe that this is unnecessarily conservative: it is secure to set the mask for Packed Load according to the target address. Doing so simplifies the operations that extract the desired value from the returned vector afterwards.

- An additional noteworthy observation regarding the CT guarantees of AVX-512 is that multiple memory load and store operations can be consolidated into a single vector instruction, provided that the conditions $\mathfrak{G}_1$ through $\mathfrak{G}_3$ are satisfied. More specifically, it is permissible to configure the mask of Packed Load/Store, or vector indices of Gather/Scatter operations according to *a set of* target addresses, to enable parallelism. This approach enables the full utilization of AVX-512 operations in CT—an opportunity that has not been previously explored.

5.3.1 Memory Layout

To enable secure and efficient CT operations in VeCT, we modify the layout of protected memory regions by (1) aligning protected data structures to the cache line size, and (2) reserving a dummy element at the beginning of each cache line. This reserved element serves as a fallback access target to ensure CT guarantees of AVX-512 operations. Consider an array of 32-bit integers. Its modified layout contains a dummy slot at the first position of each 64-byte cache line, with other array elements shifted accordingly, as illustrated in Figure 8. VeCT performs memory layout transformation consistently on all protected memory regions, including the global memory area, the heap, and the stack.

5.3.2 Constant-Time Load

In VeCT, `ct_load` is realized through two strategies based on Packed Load and Gather operations, respectively. The **Packed Load variant** allows mixed masks, including those that are secret-dependent, due to CT guarantee $\mathfrak{G}_1$. Compared with Constantine+ that always uses all-one masks, activating only the bits corresponding to the target element ensures only the intended value is retrieved, removing mask selection operations and simplifying a reduction operation afterwards, which extracts the desired value from vectors.

The **Gather variant** enforces CT behavior by assigning each index in the vector indices sequentially to a distinct adjacent cache line and applying all-one masks. Due to CT guarantee $\mathfrak{G}_3$, the implementation is secure as exactly one set bit per cache line is active. The target element is then isolated via mask selection and reduction operations.

5.3.3 Constant-Time Store

Packed Store-based. For each cache line, VeCT sets the mask bits for both the dummy element and the target element,

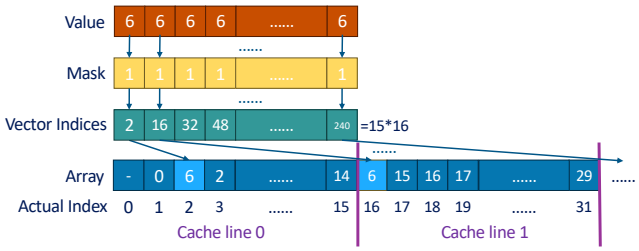


Figure 10: Scatter-based CT Store in VeCT.

if any, to be one; all other bits are set to zero. Consider the example shown in Figure 9 where the original code writes value 6 to the first element in the array.[3] In round 0, VeCT sets the first and third bits to one (i.e., with mask 0x0005) and writes value 6 to the target address. In subsequent rounds, VeCT only sets the first bit to one (i.e., with mask 0x0001) as no further target address exists. Since only the dummy element's bit is active, no other elements are modified in each cache line. More concretely, the comparison between Constantine+ and VeCT for each round is shown below:

```
//ct_store in Constantine+
o = packed_load(p, 0xFFFF)
n = mask_move(v, mask, o)
packed_store(p, n, 0xFFFF)
```

```
//ct_store in VeCT
mask = get_mask_by_compare
     (p, dst) | 0b1
packed_store(p, v, mask)
```

To the right, `get_mask_by_compare` compares the target address `dst` against the addresses that reside in the cache line starting from address `p`. It generates a `mask` where the bit corresponding to `dst` is one, if found, while all other bits are zero. So, the bit operation `or` at Line 2 correctly sets `mask`'s value as described above. Altogether, VeCT upholds CT guarantee $\mathfrak{G}2$ while eliminating preemptive loads at Line 2 in Constantine+, which improves performance (Section 6).

Scatter-based. VeCT utilizes all-one masks for scatters but sets each vector index to either the target element, or a dummy element if no target element exists for the current cache line. Figure 10 illustrates how to achieve the same functionality as Figure 9. Recall that each scatter instruction can access up to 16 cache lines (8 for 64-bit integers) in parallel. Since the target element is located at the actual index 2 within cache line 0, the first element of vector indices is set to 2, and the rest are set to the indices of dummy elements in corresponding cache lines (e.g., index 16, 32, etc.). The following code contrasts the approach taken by Constantine+ with that of VeCT:

```
//ct_store in Constantine+
o = gather(p, idx, 0xFFFF)
n = mask_move(v, mask, o)
scatter(p, n, idx, 0xFFFF)
```

```
//ct_store in VeCT
idx = generate_indices(p,
      dst)
scatter(p, v, idx, 0xFFFF)
```

On the right side, `generate_indices` first computes the specific index `i` of `dst` (relative to the base address `p`) and identifies its cache line number `j`. It then updates the (`j`+1)-th entry of the default vector indices to `i` to get `idx`. In this way, VeCT avoids preemptive loads with `gather` at Line 2 in Constantine+, while ensuring CT properties.

5.4 Vectorized Constant-time Load and Store

Programs frequently execute multiple independent and contiguous load or store operations on the same object. A classic example is accessing lookup tables such as S-boxes in cryptographic algorithm implementations, where each access is independent yet vulnerable to timing attacks. Consider a code snippet from AES algorithm in BearSSL [28], where `v0` is computed by performing four S-box lookups from the same table with different *secret-dependent* indices.

```
v0 = SboxExt0( s0 >> 24        )
   ^ SboxExt1((s1 >> 16) & 0xFF)
   ^ SboxExt2((s2 >> 8)  & 0xFF)
   ^ SboxExt3( s3        & 0xFF);
```

If each lookup is rewritten to a standalone call to `ct_load` interface, the entire table must be visited four times in its CT equivalent, which is expensive in both execution latency and memory bandwidth. Nevertheless, based on CT guarantees $\mathfrak{G}1$ and $\mathfrak{G}3$, it is possible to use one single Packed Load or Gather operation to access *all* four indices in one shot, enhancing performance by exploiting data-level parallelism while remaining CT. Thus, these benefits motivate us to provide novel vectorized CT load and store mechanisms in VeCT.

Interfaces. We introduce two new interfaces:

```
vct_load (src_list, set, set_size);
vct_store(dst_list, value_list, set, set_size);
```

where `src_list` and `dst_list` specify the target addresses to be accessed; `value_list` provides the new values for each address in `vct_store`; and `set` and `set_size` are the same as non-vectorized CT load/store instructions, specifying memory regions that the original code might touch.

5.4.1 Vectorization Identification

VeCT adopts a simple procedure to identify continuous sequences of load or store instructions that require DFL.[4] When arithmetic or other computational instructions are encountered, VeCT continues to scan the subsequent source code for additional load/store instructions, until it encounters instructions that might compromise correctness, such as function jumps or interleaved load and store operations. The collected batch, of either load or store instructions, is then consolidated into vector-width mini-batches. VeCT then rewrites each mini-batch into one `vct_load`/`vct_store` instruction.

[3] We use positions in the original array here and index them starting from 0 unless otherwise specified.

[4] We expect that a more sophisticated identification process [20,37] could further improve the performance of our work, which we leave as future work.

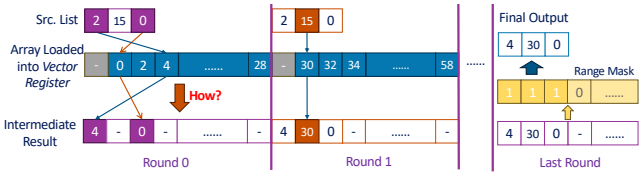


Figure 11: Packed Load-based Vectorized CT Load in VeCT.

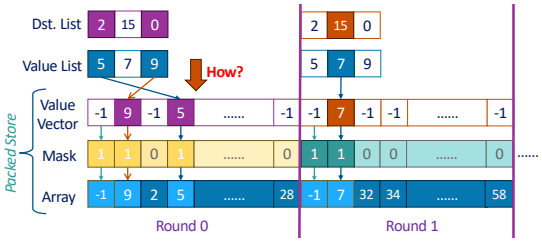


Figure 12: Packed Store-based Vectorized CT Store in VeCT.

5.4.2 Packed Load- and Store-based

We extend packed load and store instructions to perform vectorized constant-time memory accesses. However, a straightforward extension of constant-time memory accesses for a single element is not feasible in a vectorized context.

Vectorized CT Load. For a single-element load, we can simply iterate across all cache lines and extract the only value of the target address. However, vectorization introduces a challenge: each cache line may contain the data for multiple distinct target addresses simultaneously. Moreover, those target addresses are dynamic, and they can be in an arbitrary order. So one crucial issue is to *permute* the data in the loaded cache line vectors to match the order of target addresses in the program, as well as output results. Figure 11 illustrates the challenge for `vct_load`$([2,15,0],\texttt{A},680)$ where array `A` is $[0,2,4,8,...]$. A packed load for the first cache line loads a vector in the form of $[-,0,2,4,...]$ into a vector register. However, we need to place values 0 and 4 into specific designated slots in the final output, in the form of $[4,-,0,-,...]$ in an *efficient* way. Note that sorting the target addresses before invoking `vct_load` may circumvent the challenge. But the naive approach might violate CT properties. In addition, it incurs prohibitive overhead due to the extra memory accesses required for sorting.

To address this problem, `vct_load` employs a permutation-based strategy with vector extensions to efficiently rearrange loaded values in parallel and avoid time-consuming for-loops on each address in `src_list`. The process works as follows in each round: (1) it first fetches an entire cache line into a vector register using an all-one mask. (2) Then, it constructs a *permutation index vector* to specify the corresponding offset of each element within the loaded vector; the offset is marked as invalid (i.e., -1) when the element is not accessed in the current round. (3) Finally, masked permutation operations (e.g., `_mm512_mask_permutexvar_epi32/64` in AVX-512 [12]) efficiently move the relevant values from the loaded vector to their correct slots, directed by the permutation index vector. In round 0 of Figure 11, the permutation index vector is $[3,-1,1,-1,...]$ since addresses 2 and 0 in the `src_list` are located at positions 3 and 1 in the current cache line. Then a masked permutation operation with the mask `0x0005` and the permutation index $[3,-1,1,-1,...]$ copies the fourth and second elements in the loaded vector into the first and third entries in the result. This process repeats for subsequent cache lines, with the intermediate results from each round being combined to produce the final output.

Vectorized CT Store. For packed store, the challenge is similar to that of packed load, but in the reverse direction: we need to distribute values from `value_list` to potentially non-contiguous memory locations across different cache lines. Consider an example `vct_store`$([2,15,0],[5,7,9],\texttt{A},680)$ in Figure 12. To perform $\texttt{A}[0]=9$ and $\texttt{A}[2]=5$ in the first round, we need to store a vector in the form of $[-,9,-,5,...]$ with a proper mask. Therefore, the challenge is to efficiently *construct* the proper vector, where values are distributed correctly for each target address in the current round. Again, a naive sorting solution might be neither constant-time nor high-performance.

To enable efficient parallel processing with CT properties, VeCT first generates a *reverse mapping* for each round, which maps the position of each target address within the current cache line back to its original position in both `dst_list` and `value_list`; it maps other positions to an invalid position (i.e., -1). The mapping is then used to permute the values from `value_list` into a *value vector* that matches the memory layout of the current cache line. In round 0 of Figure 12, the reverse mapping is $[-1,2,-1,0,...]$ since position 0 (with address 2) and position 2 (with address 0) in `dst_list` are located at position 4 and 0 in the current cache line, respectively. Hence, the reverse mapping maps 4 to 0 and 2 to 2, respectively. A masked permutation instruction then uses the mask `0x000A` and the reverse mapping $[-1,2,-1,0,...]$ to place the values 5 and 9 into the fourth and second positions of the value vector, respectively, leaving the rest as a dummy value (-1). In the store phase, VeCT executes a packed store with a mask (e.g., `0x000B` in round 0), activating the bits for the target element(s) and the dummy element (the first one). This tactic ensures that every cache line is written to in every round, thus maintaining CT behavior. Writes to reserved elements do not alter program semantics. Across all rounds, all intended targets are updated, while in cache lines without any targets, only the reserved elements are modified.

5.4.3 Gather- and Scatter-based

Conventional gather- and scatter-based `ct_load/store` on a single element can touch up to N cache lines per round, where N is the total number of elements each gather/scatter can han-

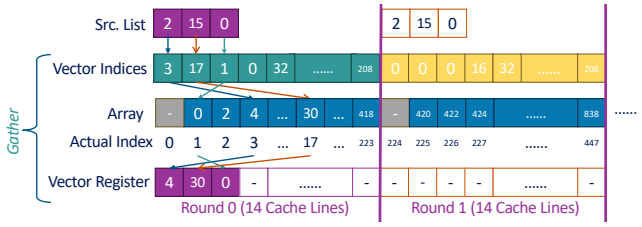


Figure 13: Gather-based Vectorized CT Load in VeCT.

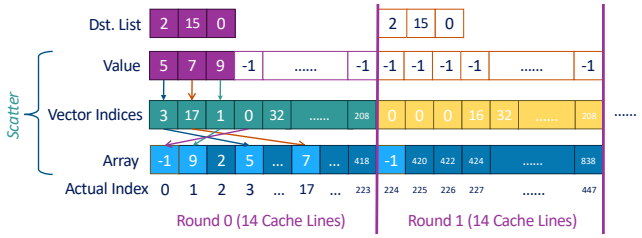


Figure 14: Scatter-based Vectorized CT Store in VeCT.

dle (e.g., 16 for 32-bit integers or 8 for 64-bit integers). In contrast, each `vct_load/store` accesses |`src/dst_list`| (i.e., the number of addresses in `src/dst_list`) elements at once. In the worst-case scenario, where all addresses in `src/dst_list` fall within a cache line, each gather/scatter can touch at most $W = N - |\texttt{src/dst\_list}| + 1$ cache lines. To mitigate this variability and ensure constant-time accesses, VeCT constrains each gather/scatter to access exactly W contiguous cache lines in every round. For example, gather can touch up to 14 cache lines in each round, as shown in Figure 13. By proactively enforcing this fixed memory footprint for each instruction, we ensure that the memory access pattern remains uniform. This approach is secure because |`src/dst_list`| is determined at compile time, and it does not change based on runtime secrets.

Vectorized CT Load. For the gather-based `vct_load`, VeCT adjusts the vector indices of gather and fixes the mask register to all-ones. In each round, the index vector is constructed as follows: (1) *Handle Target Addresses*: For each target that falls within the current round's memory range, we calculate its corresponding index. Figure 13 illustrates how `vct_load`$([2,15,0],\texttt{A},680)$ works, where array `A` is $[0,2,4,8,...]$. Since all addresses $[2,15,0]$ are within the first round's range, their corresponding indices occupy the first three slots of the vector ($[3,17,1,-,...]$). If a target is outside this range (as would be the case in later rounds), its index slot is set to a benign default value (i.e., 0). (2) *Pad with Dummy Indices*: The rest of the vector indices are assigned to point to reserved dummy elements located at fixed positions in each untouched cache line. After that, if there were still unassigned indices left in the vector indices, VeCT sets them to the default index (0) again. Note that repeated indices do not break the CT guarantee (see Section 4.3). For example, in round 0, since all three real targets have already accessed the first two cache lines, the dummy indices are crafted to access the remaining untouched lines. In the following rounds, all indices point to either dummy elements in each cache line or the default entry ($[0,0,0,0,16,32,...208]$) as the target addresses are out of the range. The intermediate results from each round are accumulated using masked selection and combination operations. Ultimately, only the values from the real targets are preserved, while all values loaded via the dummy indices are discarded. Note that gather's inherent permutation capability obviates the need to construct permutation index vectors required for packed load-based `vct_load`.

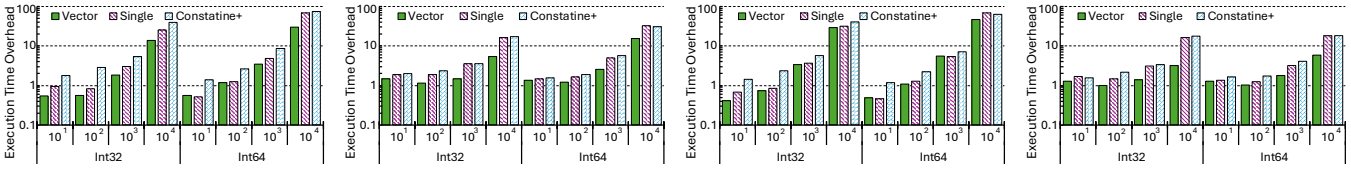
Vectorized CT Store. In this approach, VeCT maintains an all-one mask for scatter with the carefully constructed value vector and vector indices. First, a value vector is prepared. Consider the example `vct_store`$([2,15,0],[5,7,9],\texttt{A},680)$ in Figure 14. In round 0, to update addresses $[2,15,0]$, their new values are placed into the value vector sequentially from the start. All other slots are filled with a default value -1 (e.g., $[5,7,9,-1,...]$). Next, the index vector is constructed to ensure that every cache line within the current round's range receives at least one write, using the same approach as the gather-based `vct_load`. Finally, the scatter instruction is executed with an all-one mask. By construction, it writes the target values to their intended targets while storing the benign placeholder value (-1) to the dummy locations in all other cache lines, preserving constant-time security without altering the program's semantics. Likewise, scatter inherently handles the reverse mapping, eliminating extra steps required by packed store-based `vct_store`.

6 Implementation and Evaluation

We implemented VeCT on top of LLVM 9.0 [21] and Constantine [4]. For a fair comparison with Constantine+, we reused Constantine's existing framework for information flow analysis and CFL. As shown in Figure 7, VeCT performs CFL before DFL. Besides optimizing `ct_load` and `ct_store`, as elaborated in Section 5.3, VeCT also supports vectorized CT load/store operations `vct_load` and `vct_store` (Section 5.4), which are absent in Constantine+.

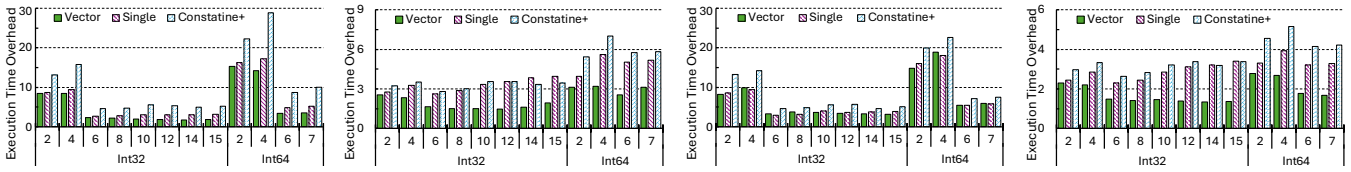
While VeCT uses AVX-512 masked non-memory instructions (e.g., selection, calculation), the latency of these instructions is independent of the mask's value, as documented by Intel [11]. Hence, by design, code generated by VeCT executes in constant time, preserving the overall security guarantee.

To justify the efficiency of VeCT, we performed experiments on a machine with an Intel Core i9-11900 and 32 GB DRAM running Ubuntu 24.04. Two variants of VeCT were evaluated, including a vectorized version (`vct_load/store`) and a single-element version (`ct_load/store`), denoted as `Vector` and `Single`, respectively. We also compared them with Constantine+. All transformed functions were executed 10,000 times after warming up in each test. The reported val-



(a) Scatter-based CT Store. (b) Gather-based CT Load. (c) Packed Store-based CT Store. (d) Packed Load-based CT Load.

Figure 15: The Execution Time Overhead (Log Scale) of Microbenchmarks with Varying Array Sizes for `Int32` and `Int64`.



(a) Scatter-based CT Store. (b) Gather-based CT Load. (c) Packed Store-based CT Store. (d) Packed Load-based CT Load.

Figure 16: The Execution Time Overhead of Microbenchmarks with Varying Access Counts for `Int32` and `Int64`.

ues correspond to the median over 10 runs. The main metric is the execution time overhead over the original insecure code for each execution. A shorter time means lower overhead.

6.1 Microbenchmark

We evaluated VeCT with microbenchmarks that perform multiple consecutive reads/writes to randomly selected positions within the same array. The access count refers to the number of consecutive accesses in each test.

Overview. We varied the array size from 10 to 10,000 elements and considered 32-bit integer (`int32`) with an access count of 12 and 64-bit integer (`int64`) data types with an access count of 6. Results are summarized in Figure 15. Across all DFL strategies, an increase in the *array size* invariably leads to higher overhead, as it increases both the frequency and volume of memory traffic. Compared to Constantine+, our `Vector` shows a significant performance advantage. For stores, `Vector` reduces overhead by 22.6% to 80.4% across both scatter- and packed store-based variants, and for loads, by 13.9% to 81.7% across both gather- and packed load-based variants. This substantial lead stems from the fact that vectorization dramatically reduces repetitive memory accesses; the entire array is touched only once per vectorized operation, rather than once per access in the original code. `Single` also reduces overhead, with `ct_store` and `ct_load` cutting overhead by up to 71.3% and 32.7%, respectively. The reduction is particularly pronounced for writes, which can be attributed to the elimination of preemptive data loads. Our approach reduces both memory traffic and the total instruction counts, avoiding performance degradation from extra cache accesses [23]. However, for a very large array (e.g., an array of size 10,000 with `int64`), the overhead reduction for `Single` is less significant because its modified memory layout introduces extra cache lines that need to be touched.

Impact of Access Count. To test the impact of access count, we varied it from 2 to 15 for `int32` and from 2 to 7 for `int64` due to the vector width in AVX-512. Its impact varies across our different strategies for `Vector` as shown in Figure 16. For the *gather-based* vectorization, the overhead reduction first increases and then decreases as the access count grows. This occurs because to maintain a constant-time pattern, each gather instruction must cover fewer cache lines at higher access counts (see Section 5.4.3), thus requiring more gather instructions overall. A similar, though less pronounced, phenomenon is observed for the *scatter-based* approach. In contrast, the *packed-load* approach achieves greater overhead reduction with larger access counts, since each packed load consistently handles one cache line, making the total number of memory accesses nearly fixed regardless of the access count. The *packed-store* approach shows a stable overhead reduction due to the additional costs of constructing reverse indices and value vectors for stores. For `Single`, its advantage narrows with more accesses. This convergence is because: (1) numerous accesses amplify the overhead of fetching additional cache lines in VeCT; and (2) cache hits and hardware prefetchers become more beneficial with repeated accesses at high access counts, narrowing the performance gap.

Security Validation. To validate the CT property of the transformed programs, we performed an additional t-test on the generated code with the access count of 2 and array size of 100. The t-test measures the timing for each program using two different secret inputs, while all other settings remain the same as Section 4.1. For all setups, the resulting t-values were below 10, indicating that the generated code is constant-time.

6.2 Real-World Applications

To further investigate performance in practical scenarios, we adopted the same benchmarks as Constantine [4] and

Table 1: The Execution Time Overhead of Real-World Applications for VeCT and Constantine+ (C+). Overhead for C+ is measured against the original insecure code, while for VeCT, results show the overhead reduction compared to Constantine+. Positive values (↓) signify an overhead reduction.

	Gather/Scatter-based			Packed Load/Store-based		
	C+	`Single`	`Vector`	C+	`Single`	`Vector`
BearSSL [8, 28]						
aes_big	27.4×	38.3% ↓	68.9% ↓	14.4×	9.1% ↓	79.9% ↓
des_tab	17.7×	-2.1% ↑	-2.0% ↑	7.3×	3.9% ↓	3.8% ↓
CHRONOS [39]						
aes	57.8×	15.0% ↓	98.8% ↓	68.3×	27.9% ↓	98.9% ↓
des	21.7×	15.4% ↓	19.3% ↓	10.7×	17.3% ↓	24.8% ↓
des3	22.2×	14.8% ↓	18.4% ↓	10.9×	16.9% ↓	24.8% ↓
anubis	33.7×	12.2% ↓	17.0% ↓	28.7×	20.1% ↓	23.4% ↓
cast5	14.4×	14.2% ↓	13.8% ↓	12.5×	27.0% ↓	26.1% ↓
cast6	70.5×	12.9% ↓	12.8% ↓	65.6×	28.2% ↓	27.9% ↓
fcrypt	17.0×	14.9% ↓	14.9% ↓	14.7×	28.4% ↓	28.4% ↓
khazad	53.7×	16.3% ↓	17.2% ↓	30.6×	25.5% ↓	28.5% ↓
APP-CR [39]						
des	21.1×	15.3% ↓	13.0% ↓	8.3×	-18.3% ↑	-17.5% ↑
RACCOON [30]						
dijkstra	54.4×	41.4% ↓	57.1% ↓	25.3×	19.5% ↓	33.8% ↓
binsearch	8.0×	20.7% ↓	21.1% ↓	8.6×	20.9% ↓	19.2% ↓
histogram	56.8×	19.0% ↓	77.5% ↓	66.6×	29.8% ↓	31.8% ↓
rsort	62.3×	1.0% ↓	1.3% ↓	56.4×	-0.2% ↑	0.0% ↓
permutation	59.5×	51.1% ↓	86.3% ↓	69.4×	37.7% ↓	12.5% ↓
heappop	7.5×	17.6% ↓	-19.5% ↑	8.1×	25.1% ↓	-64.2% ↑
LIBGCRYPT [39]						
camellia	3.6×	11.2% ↓	8.8% ↓	3.0×	13.4% ↓	17.4% ↓
des	15.6×	14.4% ↓	14.3% ↓	6.8×	16.2% ↓	15.9% ↓
seed	18.1×	12.9% ↓	13.1% ↓	15.8×	28.2% ↓	28.2% ↓
twofish	59.9×	15.3% ↓	65.8% ↓	64.4×	27.2% ↓	64.1% ↓
S-CP [39]						
aes_core	25.4×	11.1% ↓	11.5% ↓	21.6×	21.9% ↓	22.1% ↓
cast-ssl	43.3×	14.4% ↓	18.5% ↓	25.3×	26.7% ↓	28.3% ↓
PYCRYPTO [38]						
aes	56.8×	44.0% ↓	46.8% ↓	34.6×	29.2% ↓	32.9% ↓
arc4	12.5×	-17.2% ↑	34.0% ↓	14.3×	15.6% ↓	29.7% ↓
blowfish	18.2×	14.1% ↓	95.2% ↓	16.8×	31.3% ↓	94.7% ↓
cast	14.1×	13.3% ↓	13.4% ↓	13.6×	31.8% ↓	31.9% ↓
des	4.1×	8.3% ↓	19.3% ↓	2.2×	25.1% ↓	45.0% ↓
des3	5.7×	16.4% ↓	25.0% ↓	2.6×	18.8% ↓	36.0% ↓
GEO. MEAN	22.6×	17.6% ↓	46.0% ↓	16.6×	21.5% ↓	42.9% ↓

BIA [23], which are derived from various real-world applications requiring DFL. For those read-only benchmarks, we omit memory layout modification as an optimization. We measured the execution time of specific functions. For instance, in programs from cryptography libraries, we measured key generation and encryption time, and for Ghostrider, we timed their core operations. Table 1 presents the execution time overhead with VeCT and Constantine+ for each benchmark.

Overall, across all benchmarks, the geometric mean of the overhead reduction for our gather/scatter-based `Vector` and `Single` is 46.0% and 17.6%, respectively, compared to Constantine+. For our packed load/store-based versions, `Vector` and `Single` achieve overhead reductions of 42.9% and 21.5%, respectively. Taking `aes` in CHRONOS as an example, gather/scatter-based and packed load/store-based `Vectors` reduce overhead by 98.8% and 98.9%, respectively.

VeCT significantly reduces overhead in two categories of programs. **1) Write-intensive programs**: For instance, in `permutation`, the gather/scatter-based `Vector` and `Single` reduce overhead by 86.3% and 51.1%, respectively, due to the large number of assignments `a[b[i]] = i`. In contrast, for `rsort` (Radix Sort), overhead reduction is negligible. This is because the critical data structure (i.e., a small counting array) is not large enough to produce significant performance differences. **2) Programs with high vectorization potential for memory access**: As analyzed in Section 5.4, some cryptographic algorithms, including AES and Blowfish, contain a large number of parallelizable load operations, which allows VeCT to achieve marked overhead reductions. Additional characteristics for each benchmark, including the proportion of operations that can be vectorized, those requiring DFL, and the total number of memory accesses, are provided in Table 3 in Appendix B.

7 Related Work

CT Programming Analysis. Programming analyses, such as [36, 42], have been introduced to detect CT violations in software. For a comprehensive overview, we refer the reader to the recent survey presented in [9]. Other work [10, 33] focuses on studying whether optimizations used by compilers break constant-time implementations. Our work is orthogonal to those tools: VeCT automatically rewrites CT-violating code detected by CT-violation detection tools.

CT Programming Rewriter. Prior work investigates how to automatically transform programs into CT equivalents [4, 22, 34, 35, 39]. SC-Eliminator [39] assumes a weaker threat model ($\mathfrak{T}_2$), and Lif [35] identifies and resolves potential memory errors in SC-Eliminator's rewritten code. In complementary lines of work, Ma et al. [22] developed a precise static analysis to compute the differential set, an important parameter for CT load/store (see Section 2.3); Soares et al. [34] introduced partial CFL to advance CFL techniques. The most relevant work is Constantine [4], which represents the current state-of-the-art in DFL. We employ it as a primary baseline for comparison with VeCT throughout the paper.

AVX-Related Attacks. There are security attacks related to AVX. Kim et al. [16] proposed an AVX-based timing side-channel attack to compromise address space layout randomization. Downfall [25] performs a transient execution attack to leak sensitive data via gather data sampling. In contrast, VeCT focuses on how to use AVX as *a defense mechanism* for timing attacks. While we also discovered a new side channel in an existing CT code rewriting tool, the timing attack analyzes the effects of AVX-512 operations on execution time, cache-level effects, and word-level conflicts.

8 Conclusion and Future Work

In this work, we have introduced VeCT, a constant-time code rewriter that leverages vector extensions for high performance.

Our novel rewriting strategy, based on a rigorous analysis of AVX-512, reduces overhead by up to 98.9% over the state-of-the-art rewriter, while maintaining CT guarantees. This analysis also revealed a new vulnerability in the prior rewriter.

For future work, we plan to extend VeCT to other vector extensions, such as ARM SVE [2], and RISC-V "V" [32]. To do so, we can reuse our t-test framework to assess the CT guarantees of VE instructions on additional hardware platforms. If guarantees are similar to those on Intel processors, VeCT can be reused as is. However, if additional dependencies are identified, VeCT can fall back to the native adaptation with all-one masks (see Section 5.2). If some dependencies are absent, VeCT can also adopt more aggressive optimizations.

We also plan to adopt more sophisticated techniques, such as alias analysis, handling specialized function calls (e.g., math functions), loop unrolling, and instruction graph transformations [14, 20, 37], to further enhance vectorization by identifying vectorization opportunities within more complex instruction patterns.

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful comments and constructive feedback. We are grateful to Danyang Zhuo for sharing Intel Xeon servers for our evaluation, and to Michael Reiter for suggesting scientifically analyzing the timing behavior of VE instructions. We also thank Chenyang Liu for early discussions about the hypothesis testing. This work has been supported by NSF grants CNS 2401182, CNS 2401496 and a gift from Intel.

Ethical Considerations

We have carefully considered the ethical implications of our research and implemented several measures to ensure our experiments and findings were handled responsibly.

We identified four primary stakeholder groups who could be impacted by our research and its publication:

- End-Users: The general public who may use software rewritten by Constantine.
- Software Developers and System Vendors: Developers of the Constantine framework.
- The Security Research Community: Our academic and industry peers who perform side-channel defense with Constantine.
- Malicious Actors (Attackers): Individuals or groups who seek to exploit vulnerabilities that are similar to our identified vulnerability of Constantine.

End-Users. Our research process had no negative impact on end-users. All experiments were conducted in a controlled, isolated laboratory environment on our own hardware. This isolation was a core part of our experimental design, ensuring that no external systems, public data, or user services were at risk. Our work not only identifies a vulnerability of Constantine, which promotes a recent patch in Constantine that fixed the issue, but also provides a new, open-source defensive tool (VeCT) that, if adopted by end-users, will lead to more secure software. Additionally, we also urge users of any software built with Constantine to check with their vendors and promptly apply the recent patch.

Software Developers and System Vendors. To avoid publicly disclosing the identified vulnerability without prior notice to the developers, we adhered strictly to the practice of responsible vulnerability disclosure. We reported the vulnerability to the Constantine developers, providing them with our proof-of-concept and all necessary details well in advance of this publication. They acknowledged the problem and have since fixed the vulnerability, following our suggestions.[5] We suggest that all developers and vendors who use Constantine update to the latest version or use our tool VeCT instead.

The Security Research Community. To ensure our results are reproducible, transparent, and fair, all real-world applications evaluated in our paper are publicly available and open-source. This allows our peers to verify our findings, build upon our work, and perform fair comparisons, thereby advancing collective knowledge. We contribute new insights into side-channel attack vectors and the efficacy of vector extensions as a mitigation, advancing the scientific field.

Malicious Actors (Attackers). The primary potential harm of this publication is that an attacker could use our analysis of the Constantine vulnerability to craft an exploit. This harm was substantially mitigated by our adherence to responsible disclosure. The "window of opportunity" for an attacker to exploit this specific finding was minimized. Moreover, our tool VeCT serves as an alternative to thwart potential attacks on similar vulnerabilities. While any security tool could theoretically be misused by attackers to probe for weaknesses, its primary utility and design are for hardening systems, not for exploitation.

By committing to a fully isolated test environment, we eliminated risks to all external stakeholders during the research phase. The decision to publish was based on a clear harm-benefit analysis. The primary potential harm was effectively mitigated through responsible vulnerability disclosure, and the primary outcome of our paper is to arm defenders and contribute positively to the ecosystem's security.

Open Science

To promote transparency and reproducibility, the source code for our tool and all benchmarks used in our experimental

[5] Please refer to https://github.com/pietroborrello/constantine/issues/6 for more details.

evaluation are publicly available at https://doi.org/10.5281/zenodo.17822446 and https://github.com/qisheng-jiang/VeCT.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 53–70, USA, 2016. USENIX Association.

[2] Arm Limited. Introduction to SVE. https://developer.arm.com/documentation/102476/0101/, February 2025.

[3] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 555–576. Springer, 2017.

[4] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 715–733, New York, NY, USA, 2021. Association for Computing Machinery.

[5] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[6] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a DSL for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–189, 2019.

[7] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, page 45–60, USA, 2009. IEEE Computer Society.

[8] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038, 2020.

[9] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1690–1704, New York, NY, USA, 2023. Association for Computing Machinery.

[10] Lukas Gerlach, Robert Pietsch, and Michael Schwarz. Do compilers break constant-time guarantees? In *Financial Cryptography and Data Security-29th International Conference, FC*, 2025.

[11] Intel Corporation. Data operand independent timing instruction set architecture (ISA) guidance. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html, February 2023.

[12] Intel Corporation. Intrinsics for integer permutation operations. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/intrinsics-for-integer-permutation-operations.html, July 2023.

[13] Intel Corporation. Intrinsics for Intel Advanced Vector Extensions 512 (Intel AVX-512) instructions. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/intrinsics-for-avx-512-instructions.html, July 2023.

[14] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, 2011.

[15] Qisheng Jiang and Chundong Wang. Sync+Sync: A covert channel built on fsync with storage. In *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA, August 2024. USENIX Association.

[16] Suryeon Kim, Seungwon Shin, and Hyunwoo Choi. AVX-TSCHA: Leaking information through AVX extensions in commercial processors. *Computers & Security*, 134(C), November 2023.

[17] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[18] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[19] Libtom Projects. Libtomcrypt. https://github.com/libtom/libtomcrypt, May 2025.

[20] LLVM Project. Auto-Vectorization in LLVM. https://llvm.org/docs/Vectorizers.html, July 2025.

[21] LLVM Project. The LLVM compiler infrastructure. https://llvm.org/, July 2025.

[22] Cong Ma, Dinghao Wu, Gang Tan, Mahmut Taylan Kandemir, and Danfeng Zhang. Quantifying and mitigating cache side channel leakage with differential set. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[23] Yuanqing Miao, Mahmut Taylan Kandemir, Danfeng Zhang, Yingtian Zhang, Gang Tan, and Dinghao Wu. Hardware support for constant-time programming. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 856–870, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations. *Int. J. Parallel Program.*, 47(4):538–570, August 2019.

[25] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7179–7193, Anaheim, CA, August 2023. USENIX Association.

[26] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, page 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[27] Colin Percival. Cache missing for fun and profit, 2005.

[28] Thomas Pornin. BearSSL: a smaller SSL/TLS library. https://www.bearssl.org/, June 2024.

[29] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2906–2920, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: closing digital side-channels through obfuscated execution. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 431–446, USA, 2015. USENIX Association.

[31] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, page 1701–1706, Leuven, BEL, 2017. European Design and Automation Association.

[32] RISC-V. RISC-V"V" Vector Extension. https://github.com/riscvarchive/riscv-v-spec/blob/master/v-spec.adoc, January 2024.

[33] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. Breaking bad: How compilers break constant-time implementations. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '25, page 1690–1706, New York, NY, USA, 2025. Association for Computing Machinery.

[34] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. Side-channel elimination via partial control-flow linearization. *ACM Trans. Program. Lang. Syst.*, 45(2), June 2023.

[35] Luigi Soares, Fernando Magno, and Quintão Pereira. Memory-safe elimination of side channels. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 200–210. IEEE Press, 2021.

[36] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1492–1504, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. Autovesk: Automatic vectorized code generation from unstructured static kernels using graph transformations. *ACM Trans. Archit. Code Optim.*, 21(1), December 2023.

[38] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA–differential address trace analysis: finding address-based side-channels in binaries. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 603–620, USA, 2018. USENIX Association.

[39] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 15–26, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[41] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time

RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[42] Quan Zhou, Sixuan Dang, and Danfeng Zhang. CtChecker: A precise, sound and efficient static analysis for constant-time programming. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 46:1–46:26, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

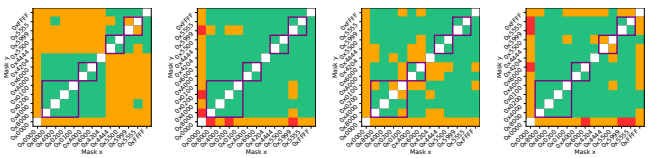
A Supplementary Results for CT Guarantees on AVX-512 Memory Access Operations

A.1 Mask Register

A.1.1 Single Cache Line

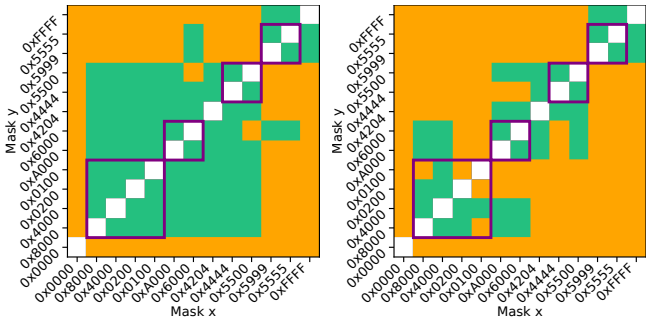
Gather. For masked gather operations, the t-test results shown in Figures 17a and 17b indicate that the time distributions are affected by both the number of 1-bits and the positions of active elements in the mask. Similarly, t-statistic becomes particularly high when comparing all-zero masks and non-zero masks. This suggests that even within a single cache line, gather instructions may incur observable side effects based on the access pattern, including differences in execution time and post-execution cache line state.

Scatter. Masked scatter instructions show similar timing variation in Figures 17c and 17d. For example, when the mask changes from all zeros to all ones, the resulting t-statistic for flush time reaches 188, suggesting that it might not be constant-time.



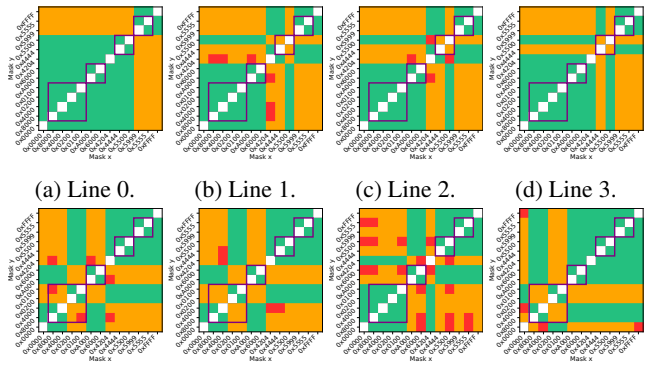
(a) Gather: Execution. (b) Gather: Cache Flush. (c) Scatter: Execution. (d) Scatter: Cache Flush.

Figure 17: T-test Results for Gather/Scatter in a Single Cache Line under Varying Mask Registers.



(a) Gather. (b) Scatter.

Figure 18: T-test Results of Execution Time for Gather/Scatter in Multiple Cache Lines under Varying Mask Registers.



(a) Line 0. (b) Line 1. (c) Line 2. (d) Line 3.

(e) Line 4. (f) Line 5. (g) Line 6. (h) Line 7.

Figure 19: T-test Results of Cache Line Flush Time for Gather in Multiple Cache Lines under Varying Mask Registers.

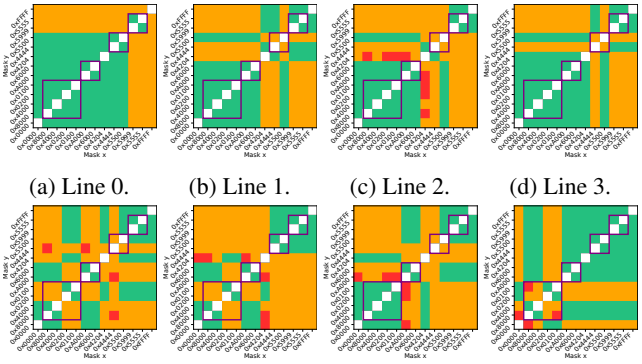
A.1.2 Multiple Cache Lines

Gather. As illustrated in Figures 18a and 19, both the number of active mask bits and their positions significantly influence the execution time and cache line flush time. This is because whether a specific cache line is accessed depends on whether any of its elements are selected for loading. For example, mask `0x0000` and mask `0x0200` both result in no access to Line 7, and thus show no evident difference in the distribution of cache line flush time. In contrast, mask `0x4000` activates a gather from Line 7, resulting in a clearly increased flush time compared to mask `0x0000`, due to the line being loaded into cache. The t-statistics for such cases are much higher than others as shown in Figure 19h, indicating that gather operations leak information about which lines are accessed, depending on the active bits in the mask.

Scatter. Masked scatter instructions exhibit similar but more pronounced timing differences, as shown in Figures 18b and 20. In this case, the effect of writing to memory causes a longer cache line flush time due to the compulsory cache writeback. For instance, comparing mask `0x0000` (i.e., no write) to mask `0x8000` (i.e., writing to Line 7) results in a substantial timing difference, again due to whether the cache line is actually modified. Meanwhile, the difference between `0x0000` and `0x0200` is negligible, as both masks result in no access to the corresponding cache line.

Table 2: The Summary of Vector Indices Used in the Experiment.

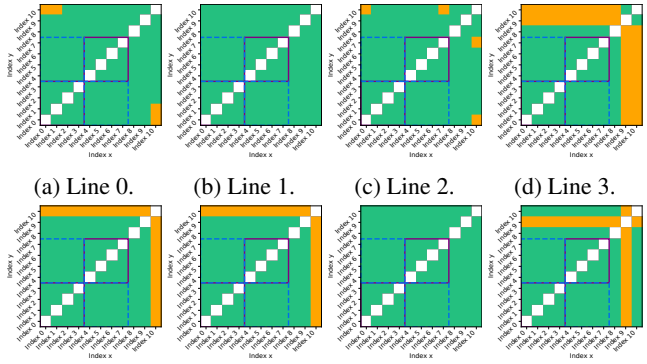
Cache Line	0	1	2	3	4	5	6	7
Index 0	0, 1	16, 17	32, 33	48, 49	64, 65	80, 81	96, 97	112, 113
Index 1	0, 1	16, 20	32, 33	48, 49	64, 65	80, 81	96, 97	112, 113
Index 2	2, 3	18, 19	34, 35	50, 51	66, 67	82, 83	98, 99	114, 115
Index 3	0, 0	16, 17	32, 32	48, 49	64, 64	80, 81	96, 96	112, 113
Index 4	0, 1, 10	17	32, 33, 40	49	64, 65, 70	81	96, 97, 100	113
Index 5	0, 1, 15	17	32, 33, 40	49	64, 65, 70	81	96, 97, 100	113
Index 6	2, 3, 12	19	34, 35, 42	51	66, 67, 72	83	98, 99, 102	115
Index 7	0, 0, 0	17	32, 32, 32	49	64, 64, 64	81	96, 96, 96	113
Index 8	0, 1, 10, 11		32, 33, 40, 41		64, 65, 70, 71		96, 97, 100, 101	
Index 9	0, 1, 2, 8, 9, 10, 11, 12				64, 65, 66, 70, 71, 72, 73, 74			
Index 10	0, 1, 2, 8, 9, 10, 11, 12						96, 97, 100, 101, 102, 103, 104, 105	



(a) Line 0. (b) Line 1. (c) Line 2. (d) Line 3.

(e) Line 4. (f) Line 5. (g) Line 6. (h) Line 7.

Figure 20: T-test Results of Cache Line Flush Time for Scatter in Multiple Cache Lines under Varying Mask Registers.



(a) Line 0. (b) Line 1. (c) Line 2. (d) Line 3.

(e) Line 4. (f) Line 5. (g) Line 6. (h) Line 7.

Figure 21: T-test Results of Cache Line Flush Time for Gather in Multiple Cache Lines under Varying Vector Indices.

A.2 Vector Indices

Table 2 summarizes the index layout used in the test. Figures 21 and 22 show the t-test results of each cache line flush time for gather and scatter in multiple cache lines under varying vector indices, respectively.

A.3 Word-Level Conflict

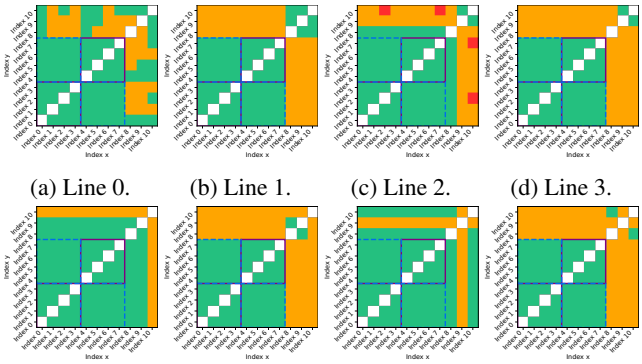
To investigate whether word-level microarchitectural conflicts, such as false dependencies or cache bank contention, are affected by the setting of mask bits in AVX-512 packed memory instructions, we followed the methodology proposed in MemJam [24] and performed the following experiments.

Packed Load and Store. A victim thread executes a masked packed load or store within a single cache line. We vary the mask to selectively enable or disable access to a specific target word. An attacker thread simultaneously runs on the sibling hyperthread of the same physical core and repeatedly accesses a group of memory addresses that may conflict with the target word. To induce potential conflict: (1) When the victim performs a **packed load**, the attacker performs a **store**; and (2) When the victim performs a **packed store**, the attacker performs a **load**. We measured the attacker's access latency in both cases and performed the hypothesis test again. Results shown in Figures 2c and 3c indicate that the mask bit would not affect the likelihood or severity of word-level conflicts for packed load and store instructions.

Gather and Scatter. A victim thread executes gather or scatter with all-one masks across multiple cache lines. We vary the vector indices (using Index 0 to 7 from Table 2) but ensure the operation always touches every target cache line. In parallel, an attacker performs read or write operations mirroring our previous test setup. As shown in Figures 4c and 5c, the results demonstrate that word-level conflicts for gather and scatter are independent of the vector indices, as long as all cache lines are touched.

A.4 Results on Intel Xeon Gold 5215

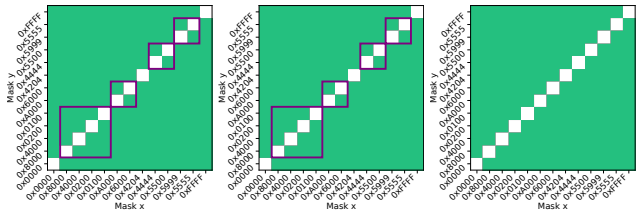
Figures 23 and 24 show the t-test results for packed load and store operations, respectively, when accessing a single cache line. Figure 25 present the t-test results for gather and scatter operations on a single cache line. Figures 26, 27, and 28 display t-test results for gather and scatter operations that access multiple cache lines. These figures analyze both execution time and cache line flush times under varying mask registers. Figures 29a, 29b, 30, and 31 also show t-test results for gather and scatter operations across multiple cache lines.



(a) Line 0. (b) Line 1. (c) Line 2. (d) Line 3.

(e) Line 4. (f) Line 5. (g) Line 6. (h) Line 7.

Figure 22: T-test Results of Cache Line Flush Time for Scatter in Multiple Cache Lines under Varying Vector Indices.



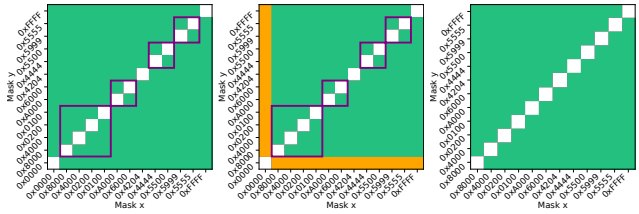
(a) Execution. (b) Cache Flush. (c) Word Access.

Figure 23: T-test Results for Packed Load in a Single Cache Line under Varying Mask Registers on Intel Xeon Gold 5215.

These figures analyze the timing effects of varying the vector indices. Figures 23c, 24c, 32a and 32b present t-test results for word-level conflicts. Figures 23c and 24c analyze packed load/store with varying masks, while Figures 32a and 32b analyze gather/scatter with varying vector indices.

B Supplementary Results for Real-World Applications

Table 3 shows the number of memory access operations that are vectorizable, those that require DFL, and the total number of access operations for each benchmark.

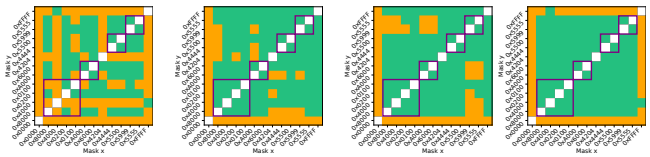


(a) Execution. (b) Cache Flush. (c) Word Access.

Figure 24: T-test Results for Packed Store in a Single Cache Line under Varying Mask Registers on Intel Xeon Gold 5215.

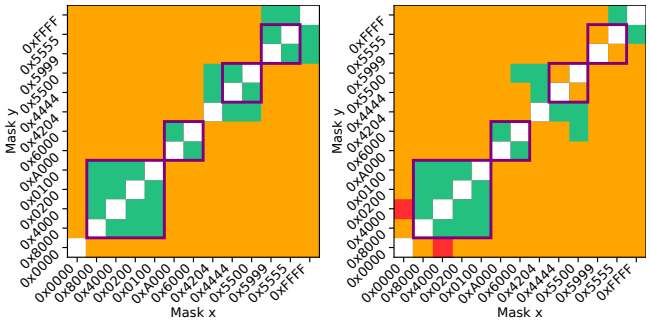
Table 3: A Comparison of the Number of Vectorized, Linearized, and Total Memory Accesses.

		Vectorized / Linearized / Total Read	Vectorized / Linearized / Total Write
BearSSL	aes_big	31/31/182	0/0/82
	des_tab	0/8/196	0/0/128
CHRONOS	aes	114/124/205	0/0/32
	des	64/206/234	0/0/39
	des3	192/525/599	0/0/100
	anubis	32/92/178	0/0/90
	cast5	0/333/391	0/0/54
	cast6	0/256/400	0/0/15
	fcrypt	0/64/94	0/0/21
	khazad	7/40/56	0/0/8
APP-CR	des	0/22/83	0/2/52
RACCOON	dijkstra	7/25/275	0/24/244
	binsearch	0/1/9	0/0/8
	histogram	8/8/17	8/8/8
	rsort	0/6/26	0/5/16
	permutation	0/1/12	8/10/12
	heappop	2/4/24	2/2/22
LIBGCRYPT	camellia	0/32/49	0/0/48
	des	0/144/199	0/0/15
	seed	2/200/277	0/0/29
	twofish	211/246/364	0/0/27
S-CP	aes_core	0/132/208	0/0/28
	cast-ssl	94/333/390	0/0/104
PYCRYPTO	aes	45/96/249	0/0/84
	arc4	2/3/11	0/2/17
	blowfish	24/24/116	0/0/75
	cast	0/284/352	0/0/85
	des	8/40/102	0/4/53
	des3	24/136/290	0/12/142



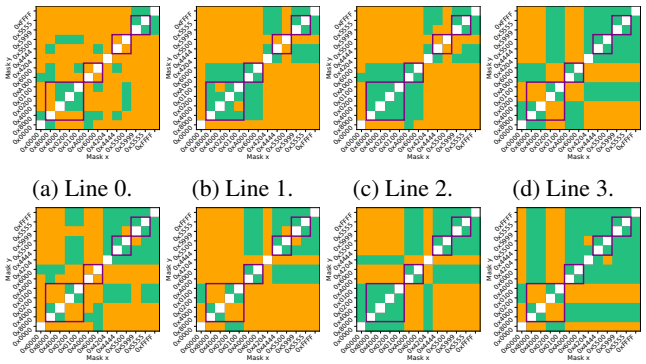
(a) Gather: Execution. (b) Gather: Cache Flush. (c) Scatter: Execution. (d) Scatter: Cache Flush.

Figure 25: T-test Results for Gather/Scatter in a Single Cache Line under Varying Mask Registers on Intel Xeon Gold 5215.



(a) Gather. (b) Scatter.

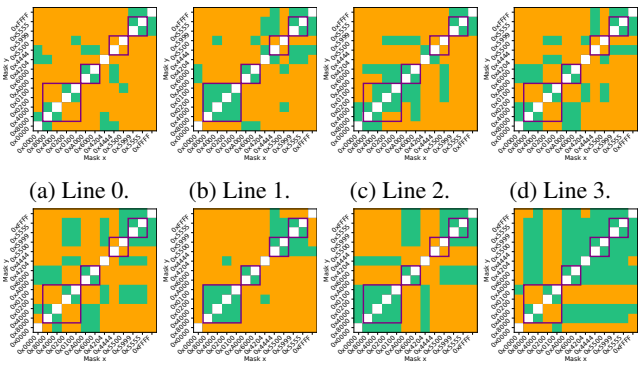
Figure 26: T-test Results of Execution Time for Gather/Scatter in Multiple Cache Lines under Varying Mask Registers on Intel Xeon Gold 5215.



(a) Line 0. (b) Line 1. (c) Line 2. (d) Line 3.

(e) Line 4. (f) Line 5. (g) Line 6. (h) Line 7.

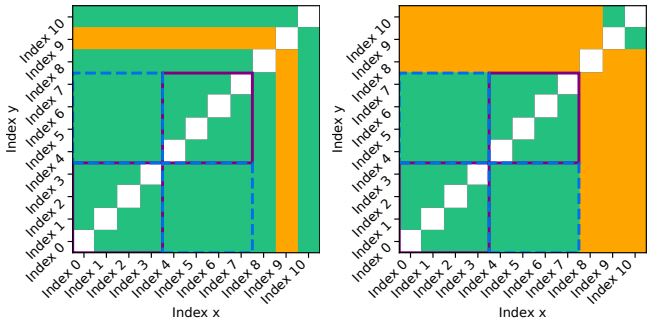
Figure 27: T-test Results of Cache Line Flush Time for Gather in Multiple Cache Lines under Varying Mask Registers.



(a) Line 0. (b) Line 1. (c) Line 2. (d) Line 3.

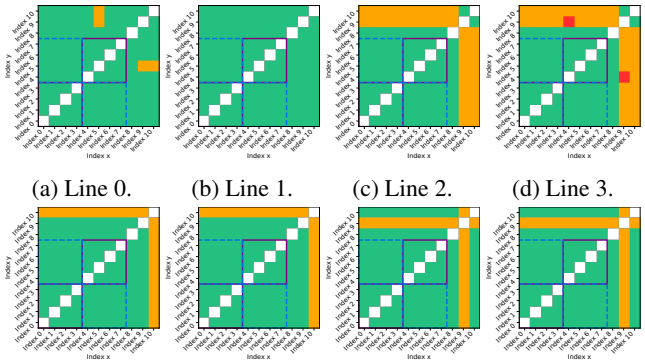
(e) Line 4. (f) Line 5. (g) Line 6. (h) Line 7.

Figure 28: T-test Results of Cache Line Flush Time for Scatter in Multiple Cache Lines under Varying Mask Registers.



(a) Gather. (b) Scatter.

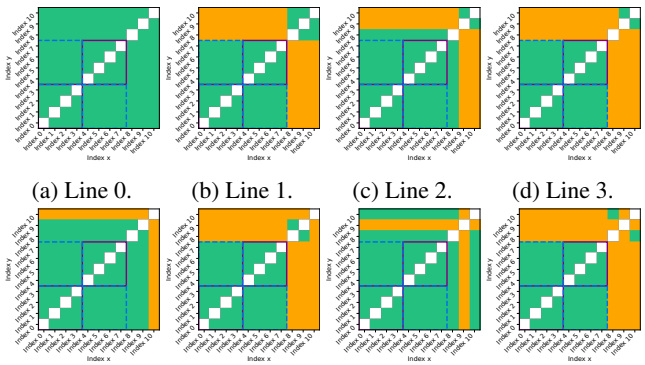
Figure 29: T-test Results of Execution Time for Gather/Scatter in Multiple Cache Lines under Varying Vector Indices on Intel Xeon Gold 5215.



(a) Line 0. (b) Line 1. (c) Line 2. (d) Line 3.

(e) Line 4. (f) Line 5. (g) Line 6. (h) Line 7.

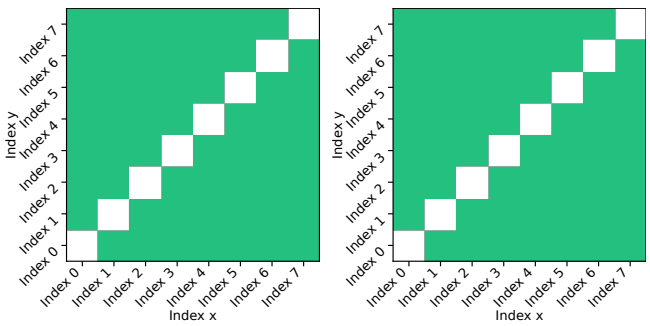
Figure 30: T-test Results of Cache Line Flush Time for Gather in Multiple Cache Lines under Varying Vector Indices on Intel Xeon Gold 5215.



(a) Line 0. (b) Line 1. (c) Line 2. (d) Line 3.

(e) Line 4. (f) Line 5. (g) Line 6. (h) Line 7.

Figure 31: T-test Results of Cache Line Flush Time for Scatter in Multiple Cache Lines under Varying Vector Indices on Intel Xeon Gold 5215.



(a) Gather. (b) Scatter.

Figure 32: T-test Results of Word-Level Conflict for Gather/Scatter in Multiple Cache Lines under Varying Vector Indices on Intel Xeon Gold 5215.