Quantifying Cache Side-Channel Leakage by Refining Set-Based Abstractions

Jacqueline L. Mitchell
University of Southern California USA

Chao Wang
University of Southern California USA

Abstract

We propose an improved abstract interpretation based method for quantifying cache side-channel leakage by addressing two key components of precision loss in existing set-based cache abstractions. Our method targets two key sources of imprecision: (1) imprecision in the abstract transfer function used to update the abstract cache state when interpreting a memory access and (2) imprecision due to the incompleteness of the set-based domain. At the center of our method are two key improvements: (1) the introduction of a new transfer function for updating the abstract cache state which carefully leverages information in the abstract state to prevent the spurious aging of memory blocks and (2) a refinement of the set-based domain based on the finite powerset construction. We show that both the new abstract transformer and the domain refinement enjoy certain enhanced precision properties. We have implemented the method and compared it against the state-of-the-art technique on a suite of benchmark programs implementing both sorting algorithms and cryptographic algorithms. The experimental results show that our method is effective in improving the precision of cache side-channel leakage quantification.

2012 ACM Subject Classification Software and its engineering → Software verification and validation; Theory of computation → Program analysis

Keywords and phrases Abstract interpretation, side-channel, leakage quantification, cache

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.13

Related Version This paper has an extended version, with appendices and extra details.
Extended Version: `https://github.com/jlmitche23/ecoop25CacheQuantification` [15]

Funding This work was partially supported by the NSF grant CCF-2220345.

1 Introduction

Cache side-channel attacks, whereby adversaries gain information about secret data by examining the footprint of program execution in the CPU cache, pose a significant threat to computer security. Cache side-channel attacks have been demonstrated in many critical infrastructure systems, ranging from cryptographic software in embedded devices [13, 30, 29, 1, 21, 19] to cloud computing applications where an adversary only needs remote access to the victim's hardware to successfully launch the attacks [5, 22, 6, 28]. Various techniques have been proposed to mitigate such attacks, including *constant-time programming* [16] along with verification techniques for proving the constant-time property [2, 4].

However, completely eliminating side-channel leakage is a challenging task since it may result in too much computational overhead [8]; it may also be infeasible for certain applications where some information leakage is required [24, 18, 27]. This motivates the development of mathematically rigorous techniques for *quantifying* side-channel leakage, to allow programmers to audit the degree of leakage in software code. While the pioneering work of Doychev et al. [12, 11] show that abstract interpretation [9] using a set-based cache abstract domain is well-suited for quantifying cache side-channel leakage, the main limitation is the loss of precision in the quantification results.

To overcome this limitation, we propose a new method for improving the precision of abstract interpretation based static program analysis for quantifying cache side-channel leakage. Static program analysis based on abstract interpretation has the advantages of being sound, generally performant, and not requiring artificially-bounded loop iterations as in unsound alternative techniques based on bounded model checking [20] or symbolic execution [7]. However, these advantages of abstract interpretation may come at the cost of precision loss. There are two key sources of precision loss in the context of cache side-channel quantification. The first source is spuriously aging memory blocks in the cache while applying the so-called *abstract transfer function* which interprets memory-accessing instructions during the analysis. It does this by taking as input an abstract cache state and returning another abstract state which overapproximates the effect of accessing a memory block on any (concrete) cache state represented by the input abstract cache state. The second source of imprecision is the spurious aging of memory blocks due to the inability to express and leverage disjunctive invariants about the abstract cache states with respect to the control flow of a program (in other words, the incompleteness of the set-based abstract domain). Our new method is designed to mitigate these two key sources of precision loss.

At the center of our method is a novel abstract transfer function for the set-based abstract domain and an automatic lifting of the domain to more accurately capture invariants about the cache state. In this work, we have applied our method in the context of the abstract domain used by CacheAudit [11], a state-of-the-art tool for quantifying cache side-channel leakage. We denote this domain as $\mathcal{C}^\sharp$. An abstract state in the $\mathcal{C}^\sharp$ domain associates each memory block with a set of possible ages, which describes the possible positions for a memory block within the cache. The ages also determine how cache states are updated under a given replacement policy, as the result of interpreting a memory-accessing instruction. During an analysis step where abstract interpretation is used to compute the resulting abstract cache state after an access to memory, we say that a memory block b is *spuriously aged* to an age a if a is in the set of possible ages for b, and yet there is no valid concrete cache state in which b is of age a in the ground truth. In some cases, applying the best abstract transformer [9], which concretizes the abstract cache state to yield a set of concrete cache states, updates each concrete cache state according to a replacement policy, and then re-abstracts the set of updated concrete states into a new abstract cache state, can mitigate spurious aging. However, even if the best abstract transformer is used, a memory block may still be spuriously aged with respect to the collecting semantics, due to the incompleteness of $\mathcal{C}^\sharp$. Consider two abstract cache states C and C' that arise due to a difference in the control flow of a program (perhaps corresponding to two different branches of an if-statement). Even if the best abstract transformer does not age b to a in both C and C', this may not be true of their abstract union; we later provide an example in Section 4. This imprecision arises due to $\mathcal{C}^\sharp$'s inability to express disjunctive invariants at the level of variations in control flow.

To address the first source of precision loss, we propose to carefully leverage information in the abstract state regarding the ages of other memory blocks when deciding to age block b, to more accurately update the abstract cache state for each memory-accessing instruction. Instead of deciding to age b only based on b's age and the age of the accessed memory block, we use the ages of all the memory blocks in the cache to prevent the spurious aging of many memory blocks. As we describe later, we prove our improved transfer function improves upon the baseline transfer function by refining it in such a way that removes cases of spurious aging.

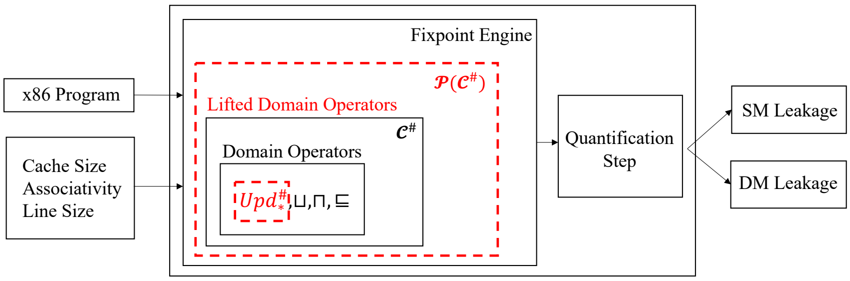

Figure 1 Our method for quantifying cache side-channel leakage based on abstract interpretation. SM corresponds to a shared memory adversary, where an attacker can observe which memory blocks are in the cache. DM corresponds to a disjoint memory adversary, where an attacker can observe which cache lines are occupied in the cache, but not the specific memory blocks occupying them.

To address the second source of precision loss, we propose to parsimoniously leverage disjunctions of abstract cache states that arise due to variations in control flow. This technique can be implemented as a refinement of $\mathcal{C}^\sharp$, based on the powerset domain introduced by [3]. The powerset domain can refine any abstract domain by lifting its operators (partial order, join, meet, widen) to operate on a lifted version of the abstract domain, whose elements are a member of the *powerset* of elements of the abstract domain.

Figure 1 shows the overall flow of our method. The input to our method consists of a program P and the cache parameters. The program P is represented in x86 binary code. The cache parameters specify the total cache size, the associativity, and the cache line size. The output of our method is the cache side-channel leakage measured in bits, for two kinds of adversaries, explained in the following. The adversary type is either Shared Memory (SM) or Disjoint Memory (DM). At the end of a program's execution, the SM adversary is able to observe the placement of memory blocks in the final cache state, along with which memory blocks are in the various locations. In contrast, the DM adversary is only able to observe which locations of the cache are occupied in the final cache state, but not the specific memory blocks that occupy them. We note that other types of adversaries are possible; we have simply chosen these adversaries to empirically evaluate our techniques. The techniques are not specific to the two adversaries in the sense that other cache analyses may also depend on such set-based abstractions. Internally, our method consists of two innovative components, shown as the new transfer function $Upd_*^\sharp$ and the lifted domain which uses disjunctions, highlighted in red, dashed boxes in Figure 1.

We have evaluated our method on a suite of 29 benchmark programs, which are implementations of various sorting algorithms and cryptographic algorithms. The baseline that we use for comparison is CacheAudit [12]. We compared the two methods on all benchmark programs, with various cache settings and adversary types. In addition to a side-by-side comparison of our method against CacheAudit, we also conducted an ablation study by enabling each of the two new techniques and then comparing the performance. The goal is to check how effective each of the two techniques is in isolation across various cache configurations, and see if they have a synergistic effect when being used together. The experimental results show that, overall, our method significantly outperforms the state-of-the-art method. Furthermore, both of the two new techniques proposed in this paper are effective, and together, they have a synergistic effect.

In summary, this paper makes the following contributions:

- We propose a new method for more accurately quantifying cache side-channel leakage based on abstract interpretation.

- We introduce two novel techniques in our method. The first leverages a new abstract transfer function to prevent spurious aging of memory blocks in the cache during the analysis. The second leverages disjunctions parsimoniously to prevent spurious aging of memory blocks due to the incompleteness of $\mathcal{C}^\sharp$.
- We prove soundness and enhanced accuracy properties of the two novel techniques.
- We implement the method and demonstrate its advantages over the state-of-the-art technique on a suite of 29 benchmark programs.

The remainder of this paper is organized as follows. After providing the technical background in Section 2, we illustrate the limitations of prior work in Section 3 using an example. Then, we present our method in Section 4 and prove the soundness and accuracy properties. We present the experimental results in Section 5. After reviewing the related work in Section 6, we give our conclusion in Section 7.

2 Background

Unlike classic program analysis techniques that focus on functional properties, e.g., control and data flows of a program, quantifying side-channel leakage also requires the modeling and analysis of non-functional properties such as the cache state. Here, we introduce the components required for abstractly modeling cache behavior.

2.1 Modeling the Cache

A cache is used to bridge the latency gap between the fast CPU and the slow main memory, to reduce the overall execution time of a program. A cache is often divided into cache sets, each of which is further divided into cache lines, where each cache line has a fixed size. Formally, a cache with the size S, the associativity n, and the line size L is organized into $m = S/(L \cdot n)$ cache sets. Each cache set consists of n cache lines. Each cache line holds a contiguous block of L bytes. Throughout the paper, let $\mathcal{B}$ refer to the set of memory blocks under consideration.

Each memory block in $\mathcal{B}$ belongs to one cache set. We define the function $set : \mathcal{B} \to \{0, \ldots, m-1\}$ that maps each memory block $b \in \mathcal{B}$ its cache set $set(b) \in \{0, \ldots, m-1\}$. Given $b_1, b_2 \in \mathcal{B}$, the condition $set(b_1) = set(b_2)$ means that the two memory blocks map to the same cache set, whereas $set(b_1) \neq set(b_2)$ means that they map to different cache sets. When $set(b_1) \neq set(b_2)$, the two memory blocks map to different cache sets, and thus do not interfere with each other.

A concrete cache state c maps each memory block in $\mathcal{B}$ to a specific age in the set $\mathcal{A} = \{0, ..., n\}$ (recall, n is the associativity of the cache). Formally, $c : \mathcal{B} \to \mathcal{A}$, where $c(b) = n$ means that the block is outside of the cache, and $0 \leq c(b) \leq n-1$ means the block is inside the cache. The ages of memory blocks are determined by the so-called *cache replacement policy*. For example, with the popular LRU (least-recently used) policy, the age of a memory block b is determined by the number of other memory blocks accessed from the last time that b was accessed during program execution.

Let $\mathcal{C}$ be the set of concrete cache states. From a concrete cache $c \in \mathcal{C}$, executing an instruction that accesses a memory block $w \in \mathcal{B}$ leads to a new cache state $Upd(c, w) \in \mathcal{C}$. Here, $Upd : \mathcal{C} \times \mathcal{B} \to \mathcal{C}$ is called the transfer function.

▶ **Definition 1.** *The transfer function* $Upd(c, w)$ *for an LRU cache state* $c \in \mathcal{C}$ *and accessed memory block* $w \in \mathcal{B}$ *is defined as follows:*

$$Upd(c, w) := \lambda b \in \mathcal{B}. \begin{cases} c(b) & \textit{when } set(b) \neq set(w) \\ c(b) & \textit{when } set(b) = set(w) \wedge b \neq w \wedge c(b) = n \\ c(b) & \textit{when } set(b) = set(w) \wedge b \neq w \wedge c(b) > c(w) \\ c(b) + 1 & \textit{when } set(b) = set(w) \wedge b \neq w \wedge c(b) < c(w) \\ 0 & \textit{when } set(b) = set(w) \wedge b = w \end{cases}$$

That is, the age of any memory block in a different cache set remains unchanged, as indicated by $set(b) \neq set(w)$. Within the same cache set, the age of the accessed memory block ($b = w$) is set to 0, the age of any memory block previously younger than the accessed block ($c(b) < c(w)$) increases by 1, and the age of any other memory block remains unchanged. In particular, $c(b) = n$ means the memory block b is already outside of the cache, and remains there upon an access to w. We note that following the LRU policy, any two memory blocks which belong to the same cache set cannot have the same age.

2.2 Abstract Interpretation of the Cache

Recall that $\mathcal{A}$ is a set of possible ages. Let $\mathcal{P}(\mathcal{A})$ be the powerset (set of all subsets) of $\mathcal{A}$, such that any element in $\mathcal{P}(\mathcal{A})$ represents a set of ages. Following Doychev et al. [11], we define the abstract cache state as a function $C : \mathcal{B} \to \mathcal{P}(\mathcal{A})$ that maps a block $b \in \mathcal{B}$ to a set of ages $C(b) \in \mathcal{P}(\mathcal{A})$. This is in contrast with the concrete state $c : \mathcal{B} \to \mathcal{A}$, which maps b to a single age $c(b)$.

Let $\mathcal{C}^\sharp$ be the set of abstract cache states. From an abstract cache state $C \in \mathcal{C}^\sharp$, executing an instruction that accesses a memory block $w \in \mathcal{B}$ leads to a new abstract cache state $Upd^\sharp(C, w) \in \mathcal{C}^\sharp$. Here, $Upd^\sharp : \mathcal{C}^\sharp \times \mathcal{B} \to \mathcal{C}^\sharp$ is called the abstract transfer function. Before defining $Upd^\sharp$, we need to define $C \downharpoonright_{w \mapsto c_w}$, which is a restriction of the abstract cache state C such that the age of block w is set to $c_w \in C(w)$. That is, $C \downharpoonright_{w \mapsto c_w}$ is an underapproximation of C where, since w occupies the age c_w, no other block can have the same age c_w, unless $c_w = n$ (meaning that w is outside of the cache), as is true in LRU caches. In the following, we define the abstract transfer function for a cache which follows the LRU replacement policy.

▶ **Definition 2.** *The abstract transfer function* $Upd^\sharp(C, w)$ *for a cache state* $C \in \mathcal{C}^\sharp$ *and accessed memory block* $w \in \mathcal{B}$ *is defined as follows [11]:*

$$Upd^\sharp(C, w) := \lambda b \in \mathcal{B}. \begin{cases} C(b) & \textit{when } set(b) \neq set(w) \\ O_n\langle w \rangle \cup O_>\langle w \rangle \cup O_<\langle w \rangle & \textit{when } set(b) = set(w) \wedge b \neq w \\ \{0\} & \textit{when } set(b) = set(w) \wedge b = w \end{cases}$$

where $O_n\langle w \rangle \cup O_>\langle w \rangle \cup O_<\langle w \rangle$ *computes a set of ages of block* $b \in \mathcal{B}$ *for each possible age* $c_w \in C(w)$*:*

- $O_n\langle w \rangle := \bigcup_{c_w \in C(w)} \{c_b \mid c_b = n \wedge c_b \in C \downharpoonright_{w \mapsto c_w} (b)\}$ *has the ages equal to* n*,*
- $O_>\langle w \rangle := \bigcup_{c_w \in C(w)} \{c_b \mid c_b > c_w \wedge c_b \in C \downharpoonright_{w \mapsto c_w} (b)\}$ *has the ages older than* c_w*,*
- $O_<\langle w \rangle := \bigcup_{c_w \in C(w)} \{c_b + 1 \mid c_b < c_w \wedge c_b \in C \downharpoonright_{w \mapsto c_w} (b)\}$ *increments ages younger than* c_w*.*

The sets $O_n\langle w \rangle$, $O_>\langle w \rangle$ and $O_<\langle w \rangle$ in Definition 2 directly correspond to the three cases $c(b) = n$, $c(b) > c(w)$ and $c(b) < c(w)$ in Definition 1.

Abstract Domain ($\mathcal{C}^\sharp$): The universe is the set of abstract cache states. Element $\top$ (top) is a state $C \in \mathcal{C}^\sharp$ such that $\forall b \in \mathcal{B} \,.\, C(b) = \mathcal{A}$. Element $\bot$ (bottom) is a state $C \in \mathcal{C}^\sharp$ such that $\forall b \in \mathcal{B} \,.\, C(b) = \{\}$.

Partial Order ($\sqsubseteq_{\mathcal{C}^\sharp}$): Given two abstract cache states $C, C' \in \mathcal{C}^\sharp$, the ordering relation $C \sqsubseteq_{\mathcal{C}^\sharp} C'$ holds if and only if $\forall b \in \mathcal{B} \,.\, C(b) \subseteq C'(b)$.

Join ($\sqcup_{\mathcal{C}^\sharp}$): Given two abstract cache states $C, C' \in \mathcal{C}^\sharp$, the join is defined as $C \sqcup_{\mathcal{C}^\sharp} C' := \lambda b \in \mathcal{B} \,.\, C(b) \cup C'(b)$.

Meet ($\sqcap_{\mathcal{C}^\sharp}$): Given two abstract cache states $C, C' \in \mathcal{C}^\sharp$, the meet is defined as $C \sqcap_{\mathcal{C}^\sharp} C' := \lambda b \in \mathcal{B} \,.\, C(b) \cap C'(b)$.

Figure 2 The abstract domain $\mathcal{C}^\sharp$ and its partial order, join, and meet operators.

For example, consider $C = \{a \mapsto \{0,1\}, b \mapsto \{1,4\}, c \mapsto \{0,2,4\}\}$, accessed memory block b, and $n = 4$. We have $C \downharpoonright_{b \mapsto 1} := \{a \mapsto \{0\}, b \mapsto \{1\}, c \mapsto \{0,2,4\}\}$ because, when the age of b is 1, the age of a can no longer be 1. However, $C \downharpoonright_{b \mapsto 4} := \{a \mapsto \{0,1\}, b \mapsto \{4\}, c \mapsto \{0,2,4\}\}$ because multiple blocks can have the age 4 (meaning they are outside of the cache). Finally, $Upd^\sharp(C, b)$ returns the abstract cache state $\{a \mapsto \{1,2\}, b \mapsto \{0\}, c \mapsto \{1,2,4\}\}$.

2.3 The Baseline Algorithm

The baseline algorithm for quantifying cache side-channel leakage using abstract interpretation consists of two steps. The *analysis step* uses the abstract transfer function to compute an abstract cache state at each program location, to overapproximate the set of concrete cache states at that location. The *quantification step* leverages the abstract cache state C at the program exit point to compute the total number of concrete cache states, which is an upper bound of the information leakage (measured in bits).

The Analysis Step: An iterative procedure using abstract interpretation and the domain operations of $\mathcal{C}^\sharp$ is used to compute an abstract cache state at each program location. The procedure assumes that all memory blocks are outside of the cache initially, i.e., $\forall b \in \mathcal{B} \,.\, C(b) = \{n\}$. Then, it applies the abstract transfer function to the abstract cache state C at each program location to compute a new abstract cache state C'. Then, it conducts standard fixpoint iteration with the abstract transfer function and the domain operations. Fixpoint iteration is required to ensure that the abstract cache computed for each program location is an invariant, i.e., that it soundly overapproximates the set of possible concrete cache states at a given program location.

Figure 2 shows the abstract domain $\mathcal{C}^\sharp$ and its partial order ($\sqsubseteq_{\mathcal{C}^\sharp}$), join ($\sqcup_{\mathcal{C}^\sharp}$) and meet ($\sqcap_{\mathcal{C}^\sharp}$) operators. Consider abstract cache states $C_1, C_2, C_3, C_4 \in \mathcal{C}^\sharp$ as examples. If $C_1 = \{a \mapsto \{0,1\}, b \mapsto \{1,2\}\}$ and $C_2 = \{a \mapsto \{0,1,4\}, b \mapsto \{1,4\}\}$, then $C_1 \sqcup_{\mathcal{C}^\sharp} C_2 = \{a \mapsto \{0,1,4\}, b \mapsto \{1,2,4\}\}$ and $C_1 \sqcap_{\mathcal{C}^\sharp} C_2 = \{a \mapsto \{0,1\}, b \mapsto \{1\}\}$. However, if $C_3 = \{a \mapsto \{0\}, b \mapsto \{1\}\}$ and $C_4 = \{a \mapsto \{1\}, b \mapsto \{0\}\}$, then $C_3 \sqcap_{\mathcal{C}^\sharp} C_4 = \{a \mapsto \{\}, b \mapsto \{\}\}$, which equals the bottom element of $\mathcal{C}^\sharp$, $\bot$. The domain operations are used in the process of fixpoint iteration. For instance, when control flow paths in the program merge, the analysis must combine abstract states using the join operator ($\sqcup_{\mathcal{C}^\sharp}$), to remain a conservative overapproximation of the true set of cache states. Furthermore, the partial order $\sqsubseteq_{\mathcal{C}^\sharp}$ is used to detect if a fixpoint has been reached.

The Quantification Step: The abstract cache state C at the program exit point is used to compute the number of concrete cache states. This is accomplished by first mapping C from the abstract domain $\mathcal{C}^\sharp$ to the concrete domain $\mathcal{P}(\mathcal{C})$. Let $\gamma_{\mathcal{C}^\sharp}$ be the concretization function, and $\gamma_{\mathcal{C}^\sharp}(C)$ be the set of concrete cache states. The cardinality $|\gamma_{\mathcal{C}^\sharp}(C)|$ represents the number of concrete cache states. In this case, $log_2|\gamma_{\mathcal{C}^\sharp}(C)|$ represents the maximum amount of information leakage measured in bits, according to Shannon's information theory.[1] We note that in our work, we assume that the leakage of each bit is equally valuable to the attacker, which motivates our use of Shannon entropy, as in CacheAudit [11].

▶ **Definition 3.** *The concretization function $\gamma_{\mathcal{C}^\sharp} : \mathcal{C}^\sharp \to \mathcal{P}(\mathcal{C})$ computes the set $\gamma_{\mathcal{C}^\sharp}(C)$ of concrete cache states for the abstract cache state C as follows: $\gamma_{\mathcal{C}^\sharp}(C) :=$*

$$\begin{aligned}\{c \in \mathcal{C} \mid\ & \forall b \in \mathcal{B} : c(b) \in C(b)\ \wedge \\ & \forall b_1, b_2 \in \mathcal{B} : set(b_1) = set(b_2) \wedge b_1 \neq b_2 \implies c(b_1) \neq c(b_2) \vee c(b_1) = c(b_2) = n\ \wedge \\ & \forall b_1 \in \mathcal{B} : 0 < c(b_1) < n \implies \exists b_2 \in \mathcal{B}.\ set(b_1) = set(b_2) \wedge (b_1 \neq b_2) \wedge c(b_2) = c(b_1) - 1\}\end{aligned}$$

The first condition $\forall b \in \mathcal{B} : c(b) \in C(b)$ takes the Cartesian product of the set of possible ages for each memory block $b \in \mathcal{B}$, while the last two conditions eliminate the obviously-invalid concrete cache states, according to the following two properties of LRU caches:

- ***No-collision within each cache set:*** If a cache line (age) is assigned to a memory block, it cannot be assigned to another memory block that belongs to the same cache set. Thus, if b_1 and b_2 belong to the same cache set ($set(b_1) = set(b_2)$) and $b_1 \neq b_2$, then $c(b_1) \neq c(b_2) \vee c(b_1) = c(b_2) = n$, meaning that the two blocks are either in different cache lines (ages) or are both outside of the cache.
- ***No-gap within each cache set:*** If a younger cache line (age) is available, an older cache line cannot be assigned to a memory block in a given cache set. Thus, when $c(b_1) \in \{1, ..., n-1\}$, there exists $b_2 \in \mathcal{B}$ such that $set(b_1) = set(b_2) \wedge (b_1 \neq b_2) \wedge c(b_2) = c(b_1) - 1$.

3 Limitations of Prior Work

While the baseline algorithm presented in Section 2 represents the state of the art, it has two main limitations in terms of the precision of its abstract transfer function and abstract domain. In this section, we use an example program to illustrate these limitations and then motivate our work on developing the new method.

3.1 The Example Program

Figure 3 shows the example program, which has a while loop containing an if-else statement. While the program has many variables, only four of them (a, b, c, and d) are being read. The two branches of the if-else statement differ in that the then-branch reads b and d whereas the else-branch reads c and d. This difference is sufficient to demonstrate the limitations of prior work and the advantages of our new method.

The Assumptions: For the sake of demonstration, we assume that all program variables in Figure 3 map to the same cache set. Furthermore, the cache set has only 4 cache lines. Finally, each variable occupies an entire cache line. With all of these assumptions, we have $\mathcal{B} = \{a, b, c, d\}$, $set(a) = set(b) = set(c) = set(d)$ and $n = 4$.

The reason why we focus only on these four variables is because, here, we assume that the cache is a *read-through, write-direct* cache as in Intel CPU's Data Direct I/O technology. That is, data is first read from main memory into the cache on a read operation, but when

[1] The Shannon entropy $H = \Sigma_c\ p(c)\ log_2 \frac{1}{p(c)}$ is maximized when each concrete cache state $c \in \gamma_{\mathcal{C}^\sharp}(C)$ has an equal probability $p(c) = \frac{1}{|\gamma_{\mathcal{C}^\sharp}(C)|}$, thus reducing H to $log_2 \frac{1}{p(c)} = log_2|\gamma_{\mathcal{C}^\sharp}(C)|$.

```c
void foo (unsigned int d) {
  unsigned int a, b, c, g, h = 0;

  d = a + d; //read {a,d}

  while (d >= 1 && d < 5) { //read {d}
    if (d < 3) { //read {d}
      h = b + 1; //read {b}
      d--; //read {d}
    }
    else {
      g = c + 1; //read {c}
      d--; //read {d}
    }
  }

}
```

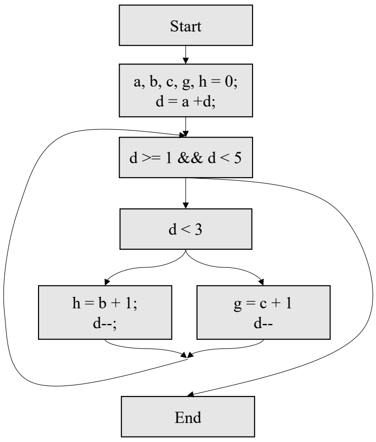

Figure 3 A program on the left-hand side and its control flow graph on the right-hand side.

writing data, it is directly written to the main memory without first updating the cache, effectively bypassing the cache for writes and prioritizing direct access to system memory. This aims to minimize unnecessary memory accesses. Under these assumptions, only read operations change the cache state.

Ground Truth: At the end of the program execution, there are only three valid concrete cache states: $c_1 = \{a \mapsto 1, b \mapsto 4, c \mapsto 4, d \mapsto 0\}$, $c_2 = \{a \mapsto 3, b \mapsto 1, c \mapsto 2, d \mapsto 0\}$, and $c_3 = \{a \mapsto 2, b \mapsto 1, c \mapsto 4, d \mapsto 0\}$. These three concrete cache states correspond to the following set of executions. State c_1 corresponds to the case where the body of the while loop is never entered, leaving c and b uncached (having age 4). State c_2 corresponds with executions in which the while loop is entered, and both branches of the if-statement are executed. (We note that due to the guards of both the while-loop and the if-statement, the then-branch is always executed after the else-branch when both are executed (when $d \geq 3$ at the beginning of the program), causing the cache line age of b to be younger than the cache line age of c. State c_3 corresponds to the case where only the then-branch of the if-statement is executed (when $d < 3$ at the beginning of the program), causing c to be uncached. With three possible concrete cache states, there is a maximum leakage of $\log_2(3)$ bits.

Baseline Algorithm: The abstract cache state at the last location of the program computed by the baseline algorithm in Section 2 is $C_{Last} := \{a \mapsto \{1,2,3,4\}, b \mapsto \{1,2,4\}, c \mapsto \{1,2,3,4\}, d \mapsto \{0\}\}$, which corresponds to 14 possible concrete cache states, where $|\gamma_{\mathcal{C}^\sharp}(C_{Last})| = 14$ (for reference, all 14 concrete cache states are featured in the appendix of the extended version [15]). This leads to a maximum leakage of $\log_2(14)$ bits, which is significantly higher than the ground truth $\log_2(3)$. As mentioned earlier, the baseline algorithm has two sources of imprecision, one of which is in the abstract transfer function $Upd^\sharp$ and the other is in the abstract domain $\mathcal{C}^\sharp$.

3.2 Imprecision of Abstract Transfer Function $\mathrm{Upd}^\sharp$

To see the imprecision in $Upd^\sharp$, consider the following abstract state C, which occurs prior to the third fixpoint iteration (using loop unrolling) of the while loop in Figure 3 using the baseline algorithm. That is, $C := \{a \mapsto \{2,3\}, b \mapsto \{1,2,4\}, c \mapsto \{1,2,4\}, d \mapsto \{0\}\}$.

The inverse of C, which maps from ages to memory blocks (variables), is shown in the

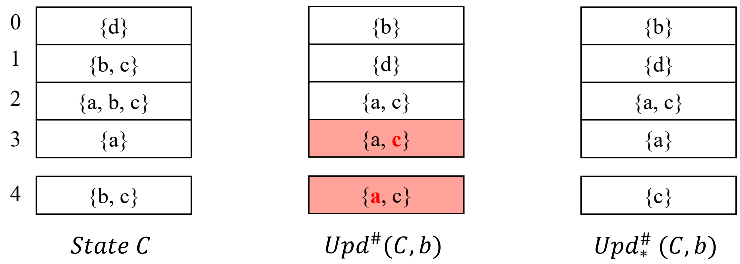

Figure 4 Differences in (baseline and new) abstract transfer functions, applied to abstract cache state $C \in \mathcal{C}^\sharp$ and the accessed memory block $b \in \mathcal{B}$.

left-most state of Figure 4.

Given abstract cache state C, consider the case of accessing (reading) variable b. As a result, the transfer function will return the abstract cache state $Upd^\sharp(C, b) := \{a \mapsto \{2, 3, 4\}, b \mapsto \{0\}, c \mapsto \{2, 3, 4\}, d \mapsto \{1\}\}$. Note that, due to spurious aging, the age 4 has become possible for a, and that the age 3 has become possible for c. However, according to the ground truth, there is no valid concrete cache state where a is outside of the cache, and there is no valid concrete cache state where c occupies the third cache line either.

In this work, we want to design a new transfer function $Upd^\sharp_*$ to eliminate such contradictory cache states. Intuitively, $Upd^\sharp_*(C, b)$ works as follows: when considering incrementing $3 \in C(a)$ to 4, $Upd^\sharp_*$ capitalizes on the fact that when a is of age 3, the set of variables with possible ages younger than 3 are $\{b, c, d\}$, as seen in the leftmost abstract cache state in Figure 4. Because there are only three such variables, and b is one of them, b must be younger than a when a is of age 3. Thus, it is unnecessary to increment $3 \in C(a)$ to 4 when accessing b, as b will already be younger than a. Similarly, $2 \in C(c)$ is not incremented to 3. Therefore, $Upd^\sharp_*(C, b) := \{a \mapsto \{2, 3\}, b \mapsto \{0\}, c \mapsto \{2, 4\}, d \mapsto \{1\}\}$, which corresponds to the rightmost abstract state in Figure 4. We shall explain more formally in Section 4.1 that, by replacing $Upd^\sharp$ with $Upd^\sharp_*$ in the iterative procedure used to analyze the program in Figure 3, our method will compute a better final abstract cache state, $C_{Last} :=\{a \mapsto \{1, 2, 3\}, b \mapsto \{1, 2, 4\}, c \mapsto \{1, 2, 4\}, d \mapsto \{0\}\}$, which corresponds to seven (instead of 14) concrete cache states at the end of program execution.

3.3 Imprecision of Abstract Domain $\mathcal{C}^\sharp$

To understand the limitation of the $\mathcal{C}^\sharp$ domain, consider the final abstract cache state $C_{Last} := \{a \mapsto \{1, 2, 3\}, b \mapsto \{1, 2, 4\}, c \mapsto \{1, 2, 4\}, d \mapsto \{0\}\}$ computed at the exit point of the example program in Figure 3 by using $Upd^\sharp_*$. As mentioned earlier, this abstract cache state corresponds to seven concrete cache states. Compared to the ground truth, which has three concrete cache states c_1, c_2 and c_3 (defined in the previous subsection), the abstract cache state C_{Last} has 4 more (spurious) concrete cache states shown below: $c_4 = \{a \mapsto 3, b \mapsto 2, c \mapsto 1, d \mapsto 0\}$, $c_5 = \{a \mapsto 2, b \mapsto 4, c \mapsto 1, d \mapsto 0\}$, $c_6 = \{a \mapsto 1, b \mapsto 4, c \mapsto 2, d \mapsto 0\}$, and $c_7 = \{a \mapsto 1, b \mapsto 2, c \mapsto 4, d \mapsto 0\}$. These spurious cache states are due to the fact that $\mathcal{C}^\sharp$ is not capable of precisely capturing disjunctive invariants that arise due to variations in control flow.

Specifically, these spurious states result from an inability of $\mathcal{C}^\sharp$ to distinguish between when the while loop is entered or not, and whether the else branch is entered at least once in the program in Figure 3. To see why, consider the final abstract cache state C_{Last}, where 1 is a possible age for a, 0 is a possible age for d, 2 is a possible age for c, and 4 is a possible age for b, thus allowing the concrete cache state $c_6 = \{a \mapsto 1, b \mapsto 4, c \mapsto 2, d \mapsto 0\}$. However, a is of age 1 only when the loop is not entered, but c being in the cache indicates that c was accessed in Line 12 of the program, and that the loop body was entered.

In this work, we want to remove these spurious states by leveraging the finite powerset framework of Bagnara et al. [3], which computes a bounded set of states (instead of a single state) at each program location. We shall explain in Section 4.2 that, in the context of cache side-channel analysis, this is accomplished by lifting the abstract domain $\mathcal{C}^\sharp$ to the powerset domain $\mathcal{P}(\mathcal{C}^\sharp)$ where each element has a cardinality of less than or equal to K. In practice, the bound K may be a small number, e.g., $K = 10$.

For the example program in Figure 3, $K = 3$ would be sufficient. That is, by using an abstract domain whose elements consist of a set of at most three elements of $\mathcal{C}^\sharp$ (as opposed to a single abstract state) to conduct fixpoint iteration with a lifted version of $\mathit{Upd}^\sharp_*$, we end up with the following abstract state: $\{\{a \mapsto \{1\}, b \mapsto \{4\}, c \mapsto \{4\}, d \mapsto \{0\}\}, \{a \mapsto \{3\}, b \mapsto \{1\}, c \mapsto \{2\}, d \mapsto \{0\}\}, \{a \mapsto \{2\}, b \mapsto \{1\}, c \mapsto \{4\}, d \mapsto \{0\}\}\}$, which corresponds to the three valid concrete cache states in the ground-truth.

We also emphasize that maintaining disjunctive invariants is able to prevent spurious aging caused by merging two abstract cache states. To see this, consider the following minor modification of code: suppose that the statement `h = g` is added between lines 8 and 9, indicating that g is read at that program location, in the then-branch of the if-statement. In the *first* iteration of analyzing the code with loop unrolling, the abstract states to be merged at the end of the if-statement are $C_{\mathit{Then}} := \{a \mapsto \{3\}, b \mapsto \{2\}, c \mapsto \{4\}, d \mapsto \{0\}, g \mapsto \{1\}\}$ and $C_{\mathit{Else}} := \{a \mapsto \{2\}, b \mapsto \{4\}, c \mapsto \{1\}, d \mapsto \{0\}, g \mapsto \{4\}\}$. Then, consider in the next iteration accessing variable b; $\mathit{Upd}^\sharp(C_{\mathit{Then}}, b) := \{a \mapsto \{3\}, b \mapsto \{0\}, c \mapsto \{4\}, d \mapsto \{1\}, g \mapsto \{2\}\}$. $\mathit{Upd}^\sharp(C_{\mathit{Else}}, b) := \{a \mapsto \{3\}, b \mapsto \{0\}, c \mapsto \{2\}, d \mapsto \{1\}, g \mapsto \{4\}\}$. Notice that in either case, a is not aged to 4. Now consider $C_{\mathit{Both}} := C_{\mathit{Then}} \sqcup_{\mathcal{C}^\sharp} C_{\mathit{Else}} = \{a \mapsto \{2, 3\}, b \mapsto \{2, 4\}, c \mapsto \{1, 4\}, d \mapsto \{0\}, g \mapsto \{1, 4\}\}$. We can see that $\mathit{Upd}^\sharp(C_{\mathit{Both}}, b)$ ages a from 3 to 4. Thus, maintaining disjunctive invariants (avoiding merging C_{Then} and C_{Else}) at this point can also prevent spurious aging. As we describe in more detail in Section 4, $\mathit{Upd}^\sharp_*$ is also unable to prevent spurious aging in this case, necessitating disjunctive invariants.

4 Our Method

We now present the two new techniques of our method for overcoming limitations of prior work. The first is a new abstract transfer function that prevents spurious aging of memory blocks in the cache. The second is a technique that lifts the abstract domain $\mathcal{C}^\sharp$ of states to *sets* of abstract cache states, to prevent spurious combinations of cache states.

4.1 The Abstract Transfer Function $\mathit{Upd}^\#_*$

Given an abstract cache state $C \in \mathcal{C}^\sharp$ and the accessed memory block $w \in \mathcal{B}$, we want to define $\mathit{Upd}^\#_*(C, w)$ such that it is significantly more accurate than the baseline $\mathit{Upd}^\sharp(C, w)$ defined in Section 2.3. Here, the focus is on eliminating contradictory cache states due to spurious aging of memory blocks, to tighten the gap between $\mathit{Upd}^\sharp$ and the *best abstract transformer* for $\mathcal{C}^\sharp$, which concretizes the abstract state C using $\gamma_{\mathcal{C}^\sharp}$, applies the concrete update function Upd to each concrete state, and abstracts the resulting set of concrete states.

4.1.1 The Intuition

To this end, recall that for the example program in Figure 3, when b is the accessed memory block and $C := \{a \mapsto \{2,3\}, b \mapsto \{1,2,4\}, c \mapsto \{1,2,4\}, d \mapsto \{0\}\}$, the spurious aging of a to 4 and the spurious aging of c to 3 will occur in $\mathit{Upd}^\sharp(C, b)$ when considering the case where b is of age 4 in C, meaning that, previously, b was outside of the cache.

Increasing the age of a from 3 to 4 is *spurious aging* because, when a is of age 3, to avoid a gap in the cache, the younger cache lines (with ages 0, 1, and 2) must hold b, c and d. Since the age of b is either 1 or 2, accessing b should not increase the age of a from 3 to 4. Increasing the age of c from 2 to 3 is also *spurious aging* because, when c is of age 2, to avoid a gap in the cache, the younger cache lines (with ages 0 and 1) must hold b and d. Since the age of b must be 1, accessing b should not increase the age of c from 2 to 3.

Leveraging the above reasoning, we want the new transfer function to return $\mathit{Upd}^\sharp_*(C, b) := \{a \mapsto \{2,3\}, b \mapsto \{0\}, c \mapsto \{2,4\}, d \mapsto \{1\}\}$. It is more accurate than $\mathit{Upd}^\sharp(C, b)$ as shown by the middle and right-most states in Figure 4 where the spurious ages in $\mathit{Upd}^\sharp(C, b)$ are highlighted in red. In fact, this is the best result that any transfer function can possibly achieve; that is, even if we concretize the abstract state C, apply $\mathit{Upd}(c, b)$ for every concrete state c, then re-abstract these concrete states, we will get the same abstract state. However, applying the aforementioned "best" abstract transformer will not be computationally efficient. In the subsections that follow, we introduce two core components of our new transfer function, $\mathit{Upd}^\sharp_*$, defined in Definition 4, to capitalize on the intuition.

4.1.2 The Function $\mathit{Var}(C, c_b, b)$

We first define $\mathit{Var} : \mathcal{C}^\sharp \times \mathcal{A} \times \mathcal{B} \to \mathcal{P}(\mathcal{B})$ as a function that takes an abstract cache state $C \in \mathcal{C}^\sharp$, an age $c_b \in \mathcal{A}$, and a memory block $b \in \mathcal{B}$ as input and returns the set of memory blocks belonging to the same cache set as b which are possibly younger than c_b in C. Formally, $\mathit{Var}(C, c_b, b) := \{b' \in \mathcal{B} \mid \exists c'_b \in C(b') \,.\, c'_b < c_b \land \mathit{set}(b) = \mathit{set}(b')\}$.

For example, consider $C := \{a \mapsto \{2,3\}, b \mapsto \{1,2,4\}, c \mapsto \{1,2,4\}, d \mapsto \{0\}\}$ (for the ease of demonstration, we assume that all memory blocks map to the same cache set). If $c_b = 3$, the set of memory blocks that are possibly younger are $\{a, b, c, d\}$; thus, we have $\mathit{Var}(C, 3, b) = \{a, b, c, d\}$. However, if $c_b = 2$, we have $\mathit{Var}(C, 2, b) = \{b, c, d\}$.

Given memory block c of age 2, we use $\mathit{Var}(C, 2, c)$ to represent the set of memory blocks (in the same cache set as c) which are possibly younger than 2 in C, and then use $\mathit{Var}(C, 2, c) \setminus \{c\}$ to remove the memory block c itself. To decide if another block b (such that $\mathit{set}(b) = \mathit{set}(c)$) may be younger than c, we check $b \in \mathit{Var}(C, 2, c) \setminus \{c\}$. For our running example, where $\mathit{Var}(C, 2, c) = \{b, c, d\}$ and $\mathit{Var}(C, 2, c) \setminus \{c\} = \{b, d\}$, the check passes, meaning that b may be younger than c (when c is of age 2).

To summarize, the above discussion shows that, in general, the condition $w \in \mathit{Var}(C, c_b, b) \setminus \{b\}$ checks if block $w \in \mathcal{B}$ may be younger than block $b \in \mathcal{B}$, when b is of age $c_b \in C(b)$ and $\mathit{set}(b) = \mathit{set}(w)$. In the next subsection, we show how to convert this "may" information into "must" information, to understand when a memory block b *must* be younger than a certain cache line age.

4.1.3 The Cardinality $|\mathit{Var}(C, c_b, b) \setminus \{b\}|$

Since $\mathit{Var}(C, c_b, b) \setminus \{b\}$ is the set of blocks in the same cache set which are younger than $b \in \mathcal{B}$, when b is of age $c_b \in C(b)$, the cardinality of the set is the number of such younger blocks. When $|\mathit{Var}(C, c_b, b) \setminus \{b\}| \le c_b$, to avoid gaps in the cache, the younger cache lines (of ages $0, \ldots, c_b - 1$) must be filled with these younger blocks. Thus, if $w \in \mathit{Var}(C, c_b, b) \setminus \{b\}$

also holds, the age of block w is younger than the age of block b, when b is of age c_b. Thus, accessing block w should not increase the age of block b when b is of age c_b.

The above condition holds in the running example when b is the accessed memory block and a is of age 3. Since $Var(C,3,a) \setminus \{a\} = \{b,c,d\}$ and $|Var(C,3,a) \setminus \{a\}| = 3$, both conditions $|Var(C,3,a) \setminus \{a\}| \leq 3$ and $b \in (Var(C,3,a) \setminus \{a\})$ hold, meaning that the age of b is younger than the age of a, when a is of age 3. Thus, accessing b should not increase the age of a, when a is of age 3. We emphasize that if the condition $|Var(C,3,a) \setminus \{a\}| \leq 3$ does not hold, i.e., $|Var(C,3,a) \setminus \{a\}| > 3$, we would not be able to ascertain that b must be younger than 3. This comes down to the “pigeon-hole” principle, where we know that if there are 4 variables for 3 possible cache lines, then b is not guaranteed to be younger than 3.

4.1.4 The Algorithm for Computing $Upd^\sharp_*(C,w)$

We define $Upd^\sharp_*(C,w)$ by revising the sets $O_>\langle w\rangle$ and $O_<\langle w\rangle$ shown in Definition 2 for $Upd^\sharp$.

▶ **Definition 4** ($Upd^\sharp_*$). *The abstract transfer function $Upd^\sharp_*(C,w)$ for cache state $C \in \mathcal{C}^\sharp$ and accessed memory block $w \in \mathcal{B}$ is defined as follows:*

$$Upd^\sharp_*(C,w) := \lambda b \in \mathcal{B}. \begin{cases} C(b) & \textit{when } set(b) \neq set(w) \\ O_n\langle w\rangle \cup O_>\langle w\rangle \cup O_<\langle w\rangle & \textit{when } set(b) = set(w) \wedge b \neq w \\ \{0\} & \textit{when } set(b) = set(w) \wedge b = w \end{cases}$$

where $O_n\langle w\rangle \cup O_>\langle w\rangle \cup O_<\langle w\rangle$ computes a set of ages of block $b \in \mathcal{B}$ for each possible age $c_w \in C(w)$:

- $O_n\langle w\rangle := \bigcup_{c_w \in C(w)}\{c_b \mid c_b = n \wedge c_b \in C\downharpoonright_{w\mapsto c_w}(b)\}$ *has the ages equal to n,*
- $O_>\langle w\rangle := \bigcup_{c_w \in C(w)}\{c_b \mid (c_b > c_w \vee (\ |\mathbf{Var(C,c_b,b)} \setminus \{\mathbf{b}\}| \leq \mathbf{c_b} \wedge \mathbf{w} \in \mathbf{Var(C,c_b,b)} \setminus \{\mathbf{b}\}\ )) \wedge c_b \in C\downharpoonright_{w\mapsto c_w}(b)\}$ *has the ages older than c_w,*
- $O_<\langle w\rangle := \bigcup_{c_w \in C(w)}\{c_b{+}1 \mid (c_b < c_w \wedge (\neg(\ |\mathbf{Var(C,c_b,b)} \setminus \{\mathbf{b}\}| \leq \mathbf{c_b} \wedge \mathbf{w} \in \mathbf{Var(C,c_b,b)} \setminus \{\mathbf{b}\})\ )) \wedge c_b \in C\downharpoonright_{w\mapsto c_w}(b)\}$ *represents the effect on ages younger than c_w.*

The sets $O_>\langle w\rangle$ and $O_<\langle w\rangle$ are revised such that, when the **highlighted** condition in $O_>\langle w\rangle$ is satisfied, we avoid incrementing the age of block b. The condition holds when the number of variables (excluding b) younger than c_b is less than or equal to c_b, and the accessed block w is one of the younger blocks. This is to prevent the spurious aging of block b.

For the example in Figure 3, in particular, the newly added conditions to $O_>\langle b\rangle$ and $O_<\langle b\rangle$ avoid the spurious aging of a from 3 to 4 and c from 2 to 3, as shown in Figure 4. Thus, by replacing $Upd^\sharp$ with $Upd^\sharp_*$ in the iterative procedure, the final abstract cache state at the end of the program in Figure 3 becomes $\{a \mapsto \{1,2,3\}, b \mapsto \{1,2,4\}, c \mapsto \{1,2,4\}, d \mapsto \{0\}\}$, which corresponds to seven (instead of 14) concrete cache states.

4.1.5 The Soundness Property

This technique is sound in that it computes an overapproximation of the concrete cache states. Recall that $Upd(c,w)$ is the concrete transfer function for a concrete cache state c and the accessed memory block w, and $\gamma_{\mathcal{C}^\sharp}$ is the concretization function. To streamline notation in the following sections, we denote Upd_w as a function which takes as input a concrete cache state, and returns the cache state after having accessed w. (This can be thought of as currying the w argument in Definition 1).

To prove soundness, we will show that $Upd^\sharp_*$ subsumes the result of the best abstract transformer. To prove this, we first explicitly define the corresponding abstraction function

$\alpha_{\mathcal{C}^\sharp} : \mathcal{P}(\mathcal{C}) \to \mathcal{C}^\sharp$, which takes as input a set of *concrete* cache states and returns an *abstract* cache state overapproximating the set.

▶ **Definition 5** ($\alpha_{\mathcal{C}^\sharp}$)**.** *Let S denote some set of concrete cache states. Then, $\alpha_{\mathcal{C}^\sharp}(S) := \lambda b \in \mathcal{B}.\{c(b) \mid c \in S\}$*

We now state the formal claim of soundness in the following theorem:

▶ **Theorem 6.** *$Upd^\sharp_*$ is sound in that, for any $w \in \mathcal{B}$ and $C \in \mathcal{C}^\sharp$, $\alpha_{\mathcal{C}^\sharp} \cdot Upd_w \cdot \gamma_{\mathcal{C}^\sharp}(C) \sqsubseteq_{\mathcal{C}^\sharp} Upd^\sharp_*(C, w)$.*

Proof. In the interest of space, we defer the full proof to the appendix of the extended version [15], and instead provide a proof sketch here. In the following, let b refer to some memory block in $\mathcal{B}$ whose ages are being updated a result of accessing memory block w.

1. We prove the soundness of $Upd^\sharp_*$ by showing that it subsumes the result of the best abstract transformer.
2. We show this by proving that if there is some concrete state c' that is the result of applying Upd_w to some state $c \in \gamma_{\mathcal{C}^\sharp}(C)$, where $c'(b) = a'$, then $a' \in Upd^\sharp_*(C, w)(b)$.
3. If $b = w$, then for all concrete states $c \in \gamma_{\mathcal{C}^\sharp}(C)$, $c(b) = 0$. It is clear to see that $0 \in Upd^\sharp_*(C, w)(b)$.
4. Otherwise, if $b \neq w$, there are three cases. First, if there is some state c, where $c(b) = n$, then $c'(b) = n$. It is clear from the definition of $Upd^\sharp_*$, that $n \in Upd^\sharp_*(C, w)(b)$ (Case $O_n\langle w\rangle$). Second, we show that if there is a concrete state c where block b is older than w, that $c(b) \in Upd^\sharp_*(C, w)(b)$ (Case $O_>\langle w\rangle$). Third, we show that if there is a concrete state c where b is younger than w, then $c(b) + 1 \in Upd^\sharp_*(C, w)(b)$ (Case $O_<\langle w\rangle$).
5. By showing that 3-4 hold, we have proved our claim.

◀

4.1.6 The Accuracy Property

We now argue that $Upd^\sharp_*$, as defined in Definition 4, is a refinement of $Upd^\sharp$, as defined in Definition 2. More formally stated, $\alpha_{\mathcal{C}^\sharp} \cdot Upd_w \cdot \gamma_{\mathcal{C}^\sharp} \sqsubseteq Upd^\sharp_*(\cdot, w) \sqsubseteq Upd^\sharp(\cdot, w)$. We now present the key theorem, describing the refinement relationship between the two transformers.

▶ **Theorem 7.** *The abstract transformer $Upd^\sharp_*$ is always more precise than, or equal to the abstract transformer $Upd^\sharp$.*

Proof. To show this, we will proceed by demonstrating that given abstract cache state C, and a memory block w to be accessed, for all $b \in \mathcal{B}$, $Upd^\sharp_*(C, w)(b) \subseteq Upd^\sharp(C, w)(b)$. We will proceed by cases.

1. **Case $\mathbf{set(b) \neq set(w)}$.** In this case, $Upd^\sharp_*(C, w)(b) = C(b)$ and $Upd^\sharp(C, w)(b) = C(b)$ by their respective definitions. Thus, $Upd^\sharp_*(C, w)(b) \subseteq Upd^\sharp(C, w)(b)$ follows immediately.
2. **Case $\mathbf{set(b) = set(w)}$.** In the case where w and b belong to the same cache set, we split up the proof into the following two cases:
   a. $\mathbf{w = b}$. In this case, memory block b is the block being accessed. Therefore $Upd^\sharp_*(C, w)(b) = \{0\}$ and $Upd^\sharp(C, w)(b) = \{0\}$, by definition. Thus, the subset relationship follows immediately.

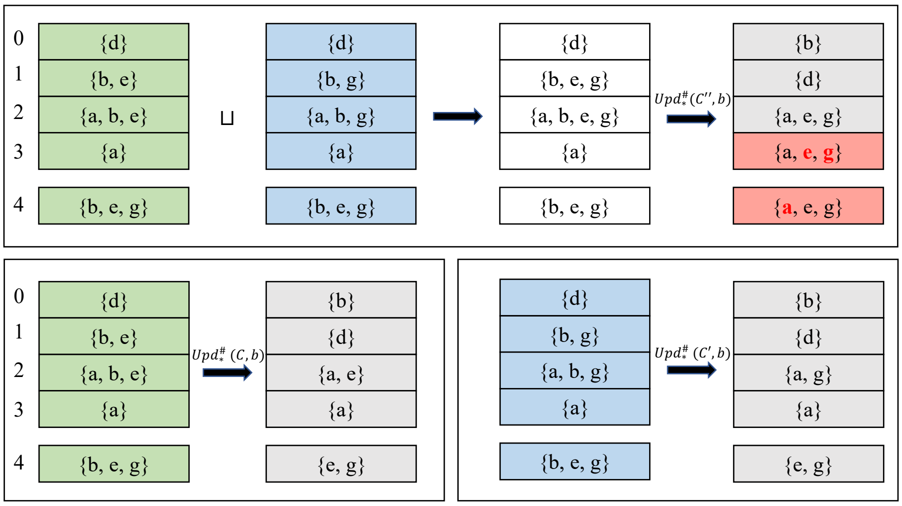

Figure 5 Merging two cache states which leads to spurious aging.

b. $\mathbf{w \neq b}$. In this case, we will use $O_n^*\langle w\rangle$, $O_>^*\langle w\rangle$, $O_<^*\langle w\rangle$, and $O_n\langle w\rangle$, $O_>\langle w\rangle$, $O_<\langle w\rangle$ to refer to the corresponding components of $Upd_*^\sharp$ and $Upd^\sharp$, respectively. For the sake of brevity, let $\mathbf{M}$ refer to the predicate $|\mathbf{Var(C, c_b, b)} \setminus \{\mathbf{b}\}| \leq \mathbf{c_b} \wedge \mathbf{w} \in \mathbf{Var(C, c_b, b)} \setminus \{\mathbf{b}\}$.

To prove the claim, we will show that $O_n^*\langle w\rangle \cup O_>^*\langle w\rangle \cup O_<^*\langle w\rangle \subseteq O_n\langle w\rangle \cup O_>\langle w\rangle \cup O_<\langle w\rangle$.

It follows by definition that $O_n^*\langle w\rangle \subseteq O_n\langle w\rangle$.

To see why $O_<^*\langle w\rangle \subseteq O_<\langle w\rangle$, it can be shown that for any c_b, c_w, that $\{c_b + 1 \mid (c_b < c_w \wedge \neg\mathbf{M}) \wedge c_b \in C\downharpoonright_{w \mapsto c_w}(b)\} \subseteq \{c_b + 1 \mid (c_b < c_w) \wedge c_b \in C\downharpoonright_{w \mapsto c_w}(b)\}$. This simply follows from the fact that $c_b < c_w \wedge \neg\mathbf{M} \implies c_b < c_w$.

Finally, it can be shown that $O_>^*\langle w\rangle \subseteq O_>\langle w\rangle$. Let $c_b \in O_>^*\langle w\rangle$. Then, either of the two conditions hold:

i. $\exists \mathbf{c_w} \,.\, \mathbf{c_b} > \mathbf{c_w}$. If $c_b > c_w$ holds, then $c_b \in O_>\langle w\rangle$, by definition of $Upd^\sharp$.

ii. $\mathbf{M}$. If $\mathbf{M}$ holds, then this implies that $w \in Var(C, c_b, b) \setminus \{b\}$. This in turn implies that there exists some c'_w such that $c'_w < c_b$. If this is the case, then $c_b \in \{c_b \mid (c_b > c'_w) \wedge c_b \in C\downharpoonright_{w \mapsto c'_w}(b)\}$. Therefore, $c_b \in O_>\langle w\rangle$, by definition.

Given the fact that $O_n^*\langle w\rangle \subseteq O_n\langle w\rangle$, $O_>^*\langle w\rangle \subseteq O_>\langle w\rangle$, and $O_<^*\langle w\rangle \subseteq O_<\langle w\rangle$, it follows that $O_n^*\langle w\rangle \cup O_>^*\langle w\rangle \cup O_<^*\langle w\rangle \subseteq O_n\langle w\rangle \cup O_>\langle w\rangle \cup O_<\langle w\rangle$.

Therefore, in any case, $Upd_*^\sharp(C, w)(b) \subseteq Upd^\sharp(C, w)(b)$. ◀

4.2 Refining the $\mathcal{C}^\sharp$ Abstract Domain

We now present the technique for extending the abstract domain to a finite powerset domain, through the framework of Bagnara et al. [3] to improve the precision of the analysis.

4.2.1 The Intuition

We first use examples to illustrate the benefit of maintaining disjunctive invariants and show how to instantiate the framework in the context of cache analysis, which leverages it to maintain a set of elements of $\mathcal{C}^\sharp$, rather than a single element of $\mathcal{C}^\sharp$.

▶ **Example 8.** As an example of why refining $\mathcal{C}^\sharp$ is useful, consider the example in Figure 5, a case where $Upd^\sharp_*$ is unable to prevent precision loss. For both the blue and green abstract cache states, when applying $Upd^\sharp_*(\cdot, b)$ on both states individually (the bottom row of the figure), we can see that it is not possible for a be age 4, nor is it possible for e or g to be age 3. However, this is not the case in their abstract join. We emphasize that applying the best abstract transformer on the joined state does not prevent this either. This indicates an imprecision of the abstract domain $\mathcal{C}^\sharp$ as opposed to sub-optimality of $\mathcal{C}^\sharp$'s operators. Thus, it is desirable to keep these two abstract states separate. More explicitly, it is desirable to have an abstract domain which maintains a set of elements of $\mathcal{C}^\sharp$, e.g. $\{C, C'\}$.

Furthermore, the refinement can be conducted when a main-channel (program values) analysis is conducted simultaneously with a side-channel (cache states) analysis. We refer to the abstract domain used in the main-channel analysis as $\mathcal{V}^\sharp$. Let $\mathcal{V}$ be some numerical abstract domain which approximates a numerical domain ($\mathcal{P}(\mathbb{Z})$) (for instance, $\mathcal{V}$ may be the domain of intervals or a domain of integer-valued sets). Let V be the set of program variables. Then, we assume $\mathcal{V}^\sharp := V \to \mathcal{V}$ is the abstract domain for the set of variables in the program which consists of maps from variables to an abstract value representation $\mathcal{V}$.

In this case, the abstract domain to be refined is the abstract domain which has elements of tuples of an abstract state in $\mathcal{V}^\sharp$ and an abstract state in $\mathcal{C}^\sharp$. The respective abstract domain operators are applied, independently, pointwise. The domain is denoted by $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$.

In fact, refinement at the level of both the program value and cache abstractions can be useful, because if the value abstractions are more precise, then certain paths in the control of the program (and subsequently, memory accesses) may be eliminated, possibly leading to abstract cache states that are more precise. We write the rest of the section with this in mind (and it corresponds to the set-up in our evaluation). Therefore, in the remainder of this section we consider the concrete domain to be $\mathcal{P}(V \to \mathbb{Z}) \times \mathcal{P}(\mathcal{C})$.

4.2.2 The Finite Powerset Domain

In this section, we introduce the finite powerset domain. We first begin by introducing relevant notation and operators.

The maximum number of abstract states of type $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$ allowed in the aforementioned set, denoted k, is pre-defined by the user. We refer to elements of such a set as disjuncts. In our case, the finite powerset framework can be thought of taking $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$, which approximates $\mathcal{P}(V \to \mathbb{Z}) \times \mathcal{P}(\mathcal{C})$, and replacing it with an abstract domain which still approximates $\mathcal{P}(V \to \mathbb{Z}) \times \mathcal{P}(\mathcal{C})$, but using a *set* of abstract values in $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$, that is, an element of the powerset of $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$, $\mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$. With a slight abuse of notation, let $\mathcal{V}^\sharp \times \mathcal{C}^\sharp := \langle \mathcal{V}^\sharp \times \mathcal{C}^\sharp, \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}, \perp_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}, \top_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}, \sqcup_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}, \sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} \rangle$ denote the abstract domain $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$, along with its operators. We say that an element $S \in \mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ is *non-redundant* with respect to $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$ if and only if $\perp_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} \notin S$ and $\forall s_1, s_2 \in S. s_1 \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s_2 \implies s_1 = s_2$. Non-redundancy ensures that a set of abstract states does not contain unnecessary elements that are already represented by other elements in the set.

A *subsumption operator* serves to normalize an element $S \in \mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ by removing redundant elements. The formal definition of the subsumption operator is in Figure 6. The $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$ operator removes redundant states based on both abstract program states and abstract cache states. The $\Omega_R^{\mathcal{C}^\sharp}$ operator merges abstract states which share the same abstract *cache* states; we emphasize that $\Omega_R^{\mathcal{C}^\sharp}$ does not remove any states based on redundancy, it simply merges abstract states which share the same abstract *cache* states. As we will see later on, the subsumption operator is used in the definition of the join, meet, and widening

Cache-Based Merging Operator $(\Omega_R^{\mathcal{C}^\sharp} : \mathcal{VC}^\sharp \to \mathcal{VC}^\sharp)$. $\Omega_R^{\mathcal{C}^\sharp}$ takes in an abstract state S and merges any two elements of S if they have the same abstract **cache** state.

Subsumption Operator $(\Omega^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} : \mathcal{VC}^\sharp \to \mathcal{VC}^\sharp)$. $\Omega^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$ takes in an abstract state S and removes elements of S if they are subsumed by some other state in S. $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S) \mapsto S'$, where $S' := S \backslash \{s \in S \mid s = \bot_{\mathcal{VC}^\sharp} \vee \exists s' \in S. s \sqsubset_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s'\}$.

Figure 6 The subsumption and merging operators with respect to $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$ and $\sqsubseteq_{\mathcal{C}^\sharp}$.

operators, while $\Omega_R^{\mathcal{C}^\sharp}$ is used in the definition of the join operator.

Let $\mathcal{P}_{fn(k)}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp, \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp})$ denote the set of all elements of $\mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ which have a cardinality of at most k. Formally, $\mathcal{P}_{fn(k)}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp, \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}) := \{S \in \mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \mid |S| \leq k\}$, where every element S is non-redundant according to $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$. With this in place, we now formally define the finite powerset domain:

► **Definition 9** (Finite Powerset Domain $\mathcal{VC}^\sharp$)**.** *Let* $\mathcal{VC}^\sharp := \langle \mathcal{P}_{fn(k)}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp, \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}), \sqsubseteq_{\mathcal{VC}^\sharp}, \bot_{\mathcal{VC}^\sharp}, \top_{\mathcal{VC}^\sharp}, \oplus_{\mathcal{VC}^\sharp}, \sqcap_{\mathcal{VC}^\sharp}\rangle$ *denote the finite powerset domain. Here,* $\bot_{\mathcal{VC}^\sharp} = \emptyset$ *and* $\top_{\mathcal{VC}^\sharp} = \{\top_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}\}$. $\sqsubseteq_{\mathcal{VC}^\sharp}$ *is defined as:* $S \sqsubseteq_{\mathcal{VC}^\sharp} S' \iff \forall s \in S : \exists s' \in S'. s \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s'$, *as in [3].* $S \oplus_{\mathcal{VC}^\sharp} S'$ *is defined to be* $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S \cup S')$.

$\mathcal{VC}^\sharp$ *is related to the concrete domain* $\mathcal{P}(V \to \mathbb{Z}) \times \mathcal{P}(\mathcal{C})$, *with the following concretization function:* $\gamma : \mathcal{VC}^\sharp \to (\mathcal{P}(V \to \mathbb{Z}) \times \mathcal{P}(\mathcal{C}))$, *where* $\gamma(S) \mapsto \bigcup\{(v, c) \in \gamma_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(s) \mid s \in S\}$.

In summary, Definition 9 states that the lifted abstract domain $\mathcal{VC}^\sharp$ consists of sets of elements of $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$, where $S \in \mathcal{VC}^\sharp$ is lower than $S' \in \mathcal{VC}^\sharp$ w.r.t. the partial order $\sqsubseteq_{\mathcal{VC}^\sharp}$ if every element in S is subsumed by some element in S', according to $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$. $\mathcal{VC}^\sharp$ relates to the concrete domain via the concretization function γ that takes an abstract element S and returns the union of the concretization of each element of S w.r.t. $\mathcal{V}^\sharp \times \mathcal{C}^\sharp$.

We now introduce each of the necessary domain operations for $\mathcal{VC}^\sharp$. We begin by introducing the lifted versions of the abstract transfer functions and the meet operator, and relegate join to its own subsection.

Abstract Transfer Functions For both instructions that impact program values as well as the abstract cache state, we lift the application of the transfer function to be elementwise. Let $s[C]$ and $s[V]$ denote the cache abstraction and value abstraction components, respectively. Let $T_{\mathcal{V}^\sharp}$ be a transfer function that affects the abstract state corresponding to the program values. Then, the lifted version for $\mathcal{VC}^\sharp$ is a function such that $S \mapsto \Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}\{(T_{\mathcal{V}^\sharp}(s[V]), s[C]) \mid s \in S\}$. Similarly, let $T_{\mathcal{C}^\sharp}$ be a transfer function that affects the part of the abstract state corresponding to the abstract cache state. Then, the lifted version for $\mathcal{VC}^\sharp$ is a function such that $S \mapsto \Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}\{(s[V], T_{\mathcal{C}^\sharp}(s[C])) \mid s \in S\}$.

The Meet Operator Meet is defined by taking the pairwise meet w.r.t. $\sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$. Specifically, if $S, S' \in \mathcal{VC}^\sharp$, then $S \sqcap_{\mathcal{VC}^\sharp} S'$ is defined as $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(\{s \sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s' \mid s \in S, s' \in S'\})$. We note that this set may be larger than k. In this case, we can view the meet operator as replacing Line 1 of the algorithm for join (Algorithm 1) with $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(\{s \sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s' \mid s \in S, s' \in S'\})$. The justification for the validity of the meet operator is in the appendix of the extended version [15].

Now, in the next section, we introduce our join operator.

4.2.3 The Join Operator

We now present the abstract **join** operator, which is the key novelty of our technique. In effect, this will replace the role of $\oplus_{\mathcal{VC}^\sharp}$ in Definition 9 to ensure that the number of disjuncts

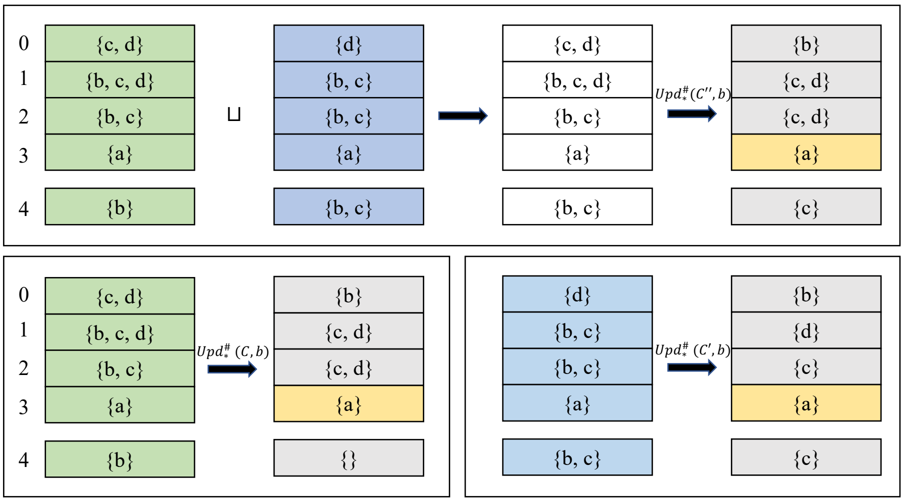

Figure 7 Merging two cache states does not cause spurious aging.

remains at most k when the join operator is applied by the analysis. Typically, deciding how to maintain and manage the disjunctive components in techniques like trace-partitioning [23], disjunctive completion [10], and the finite powerset framework is a key challenge in effectively implementing these techniques. In order to do so, we first consider when it is necessary to maintain certain disjuncts to prevent spurious aging:

▶ **Example 10.** Consider the two following abstract cache states for a fully-associative cache (all blocks map to one cache set) which can store four memory blocks (associativity = 4): $C = \{d \mapsto \{0,1\}, b \mapsto \{1,2,4\}, c \mapsto \{0,1,2\}, a \mapsto \{3\}\}$ (green) and $C' = \{d \mapsto \{0\}, b \mapsto \{1,2,4\}, c \mapsto \{1,2,4\}, a \mapsto \{3\}\}$ (blue), depicted in Figure 7. We can see that upon an access to variable b, $\mathit{Upd}^\sharp_*$ will not increment $3 \in C(a)$ to 4, meaning that a is definitely in the cache.

We can also see that this is true for their abstract join $C'' = C \sqcup_{\mathcal{C}^\sharp} C' = \{d \mapsto \{0,1\}, b \mapsto \{1,2,4\}, c \mapsto \{0,1,2,4\}, a \mapsto \{3\}\}$, where $4 \notin \mathit{Upd}^\sharp_*(C'', b)(a)$. In this example, we can see that merging the blue and green cache states did not impact the ability of $\mathit{Upd}^\sharp_*$ to prevent spuriously aging a from 3 to 4. Despite neither abstract state being subsumed by the other, the reason why $\mathit{Upd}^\sharp_*$ is able to prevent spurious aging on the union of both states is that the set of concrete cache states represented by the blue abstract state is a subset of the concrete states represented by the green state. This, combined with the fact that $\mathit{Upd}^\sharp_*$ prevents a from spuriously aging in either state, means that the same holds for the joined state.

The scenario referred to in Example 10, is a sufficient, but not necessary condition. To see why, consider another example, in which the set of valid concrete states of C and C' do not subsume one another, as in the following example:

▶ **Example 11.** Consider the two following abstract cache states: $C = \{d \mapsto \{0,1\}, b \mapsto \{1,2,4\}, c \mapsto \{0,2\}, a \mapsto \{2,3\}\}$ and $C' = \{d \mapsto \{0,1\}, b \mapsto \{0,2,4\}, c \mapsto \{1,3\}, a \mapsto \{2,3\}\}$. We can see that the set of concrete states represented by C and C' are not subsumed by one-another. (For example, $\{c \mapsto 0, d \mapsto 1, b \mapsto 2, a \mapsto 3\} \in \gamma_{\mathcal{C}^\sharp}(C)$, but is not in $\gamma_{\mathcal{C}^\sharp}(C')$ and $\{d \mapsto 0, c \mapsto 1, b \mapsto 2, a \mapsto 3\} \in \gamma_{\mathcal{C}^\sharp}(C')$, but is not in $\gamma_{\mathcal{C}^\sharp}(C)$.) However, in their abstract join, $C'' = \{d \mapsto \{0,1\}, b \mapsto \{0,1,2,4\}, c \mapsto \{0,1,2,3\}, a \mapsto \{2,3\}\}$, the following concrete state becomes possible: $\{b \mapsto 0, d \mapsto 1, c \mapsto 2, a \mapsto 3\}$, which is not a valid cache state in either C or C'. But, $\mathit{Upd}^\sharp_*(C'', b)$ is still able to avoid spuriously aging a from 3 to 4.

The scenarios described by Example 10 and Example 11 are in contrast with the example in Figure 5. In the Figure 5 example, a concrete cache state which is not possible in either cache state becomes possible in their abstraction union, and a memory block was spuriously aged as a result. In the case of Example 10 no new (valid) concrete cache state is introduced by the result of the join of the two abstract states. However, in Example 11, a new concrete cache state is introduced by the result of their join, but spurious aging was prevented. Having such a wide range of possibilities motivates the search for understanding when to merge abstract cache states, and when not to. If computational resources were no limit, only merging abstract cache states such that $\gamma_{\mathcal{C}^\sharp}$ is *distributive* over the two cache states is ideal, meaning that no infeasible concrete cache states will be introduced. However, this is not applicable in practice due to being too costly, for two reasons. The first is that the number of disjuncts needed to be maintained may be very large (perhaps infinite in certain cases, depending on the control flow of the program, taking us out of the scope of the finite powerset construction), and the second is that checking the abstract states by concretizing them each time may lead to a large computational overhead.

Therefore, instead, we aim to maintain a reasonable number of disjunctions while retaining some precision, by carefully merging abstract states. To do so, we introduce our join operator to replace $\oplus_{\mathcal{VC}^\sharp}$ in Definition 9.

The Algorithm for Join: The abstract join is parameterized by the maximum number of disjuncts allowed, as well as a similarity relation $\sim_R$, to merge states when the number of allowed disjuncts is exceeded. The similarity relation takes in two states $s, s' \in \mathcal{V}^\sharp \times \mathcal{C}^\sharp$ and returns true or false, depending on whether they should be merged.

Algorithm 1 Join Operation $\oplus^*_{\mathcal{VC}^\sharp}\langle k : \mathbb{N}^+, \sim_R : (\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \times (\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \to \{\texttt{tt}, \texttt{ff}\}\rangle$

Input: $S, S' \in \mathcal{VC}^\sharp$ // $\oplus^*_{\mathcal{VC}^\sharp} : \mathcal{VC}^\sharp \times \mathcal{VC}^\sharp \to \mathcal{VC}^\sharp$
Output: Joined state set S''
1. Set $S'' := \Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S \cup S')$
2. **if** $|S''| \le k$ **then**
  return S''
3. **else**
  a. Set $S'' := \Omega_R^{\mathcal{C}^\sharp}(S'')$
  b. **if** $|S''| \le k$ **then**
   return $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S'')$
  c. **else**
   i. **while** $|S''| > k$ **and** $\exists s_1, s_2 \in S'' : s_1 \sim_R s_2$ **do**
    Merge states s_1 and s_2 where $s_1 \sim_R s_2$
   ii. **while** $|S''| > k$ **do**
    Arbitrarily merge any pair of states
   iii. return $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S'')$

The join operator begins by combining the disjuncts in S, S' and removes redundant elements w.r.t. the value and cache abstractions. If, after doing this step, the cardinality of the resulting set is less than or equal to k, we stop. This choice is motivated by the fact that preserving disjunctive information that differs on the value domain may lead to more precise control-flow information, and thus, possibly result in fewer spurious memory accesses. Otherwise, the join operator aims to merge elements which share the same abstract cache states, using $\Omega_R^{\mathcal{C}^\sharp}$. After this, the subsumption operator is applied to ensure non-redundancy. If the number of disjuncts are within limit, the join operator returns the resulting abstract state. However, if the number of disjuncts still exceeds the number of those which are allowed, the join operator merges pairs of states which satisfy $\sim_R$. Merging via $\sim_R$ is done as much as possible until the number of disjuncts remaining are at most k. If all states satisfying $\sim_R$ have been merged pairwise and the number of disjuncts exceeds k, then states are merged arbitrarily pairwise, until the number of disjuncts is at most k. After this is complete, the subsumption operator is applied to ensure non-redundancy. We show that the join operator is valid in the appendix of the extended version [15]. In the next section, we discuss the possibilities for $\sim_R$.

4.2.4 The Merging Strategies

Recall that in Example 11, the concrete cache state $\{b \mapsto 0, d \mapsto 1, c \mapsto 2, a \mapsto 3\}$ is captured by the abstract cache state C'', but not by C or C', but we can still prevent the spurious aging of a from 3 to 4 by applying $Upd^\sharp_*$ to the abstract state C''. The key factor in $Upd^\sharp_*(C'', b)$ being able to avoid spuriously aging a from 3 to 4 is that the set of variables younger than age 3 (excluding a) is the same across C, C', C'' — the set being $\{d, b, c\}$. This corresponds with the second condition from the definition of $O_>\langle w\rangle$ in $Upd^\sharp_*$. Thus, it is of interest to preserve that property whenever we can.

To this end, we consider a condition under which merging two abstract states preserves the ability of $Upd^\sharp_*$ to prevent spurious aging of a given memory block when accessing some memory block w on the two abstract states separately. Specifically, we consider when two abstract cache states C and C' can be merged such that if $Upd^\sharp_*(C, w)$ and $Upd^\sharp_*(C', w)$ do not age a memory block b from t to $t+1$, then $Upd^\sharp_*(C \sqcup_{\mathcal{C}^\sharp} C', w)$ does not age memory block b from t to $t+1$, where t is some possible cache line age between 1 and $n-1$. If they satisfy the property that the set of memory blocks who have ages younger than t are the same in C and C', then if $Upd^\sharp_*(C, w)$ and $Upd^\sharp_*(C', w)$ do not spuriously age block b from t to $t+1$, then $Upd^\sharp_*(C'', w)$ where $C'' = C \sqcup_{\mathcal{C}^\sharp} C'$ does not age block b from t to $t+1$ spuriously.

We first introduce a helper function used in the proceeding Lemma that formalizes the aforementioned property. Let $\max_{<t}(A)$ be a function that takes a set of integers (A), and returns the maximum value that is less than t. If no such value exists, it returns $-\infty$. For example, $\max_{<4}(\{1, 2, 3, 4\})$ returns 3, while $\max_{<5}(\{5, 6, 7\})$ returns $-\infty$. We now state the Lemma:

▶ **Lemma 12.** *Let $C, C' \in \mathcal{C}^\sharp$. Let $w \in \mathcal{B}$ be the memory block being accessed. Let $t \in \{1, ..., n-1\}$ be some cache line age, where n is the associativity of the cache. If for each $b \in \mathcal{B}$, $\max_{<t}(C(b)) = \max_{<t}(C'(b))(\star)$, then, if $Upd^\sharp_*(C, w)$ and $Upd^\sharp_*(C', w)$ do not age b from t to $t+1$, then $Upd^\sharp_*(C \sqcup_{\mathcal{C}^\sharp} C', w)$ does not age b from t to $t+1$.*

Proof. For the proof, please refer to the appendix of the extended version [15]. ◀

For simplicity, Lemma 12 assumes C and C' are fully-associative, and the lemma extends naturally to caches with more than one set. The lemma yields two key corollaries for our purposes. The first states that if the universe of concrete cache states are partitioned using a set of abstract cache states based on groups of states which satisfy ($\star$) pairwise, then there will be no spurious aging of memory blocks caused by combining states with the abstract join operator. The second one states a special case of the lemma, which prevents memory blocks from being spuriously uncached as a result of combining states with the abstract join operator. (We note that there is still possible imprecision due to the gap between $Upd^\sharp_*$ and the best abstract transformer.)

▶ **Corollary 13.** *Merging abstract cache states based on the strategy of only merging abstract cache states C, C' that satisfy the following formula $\forall t \in \{1, ..., n-1\}. \forall b \in \mathcal{B}. \max_{<t}(C(b)) = \max_{<t}(C'(b))$ will result in no block being spuriously aged, purely due to combining abstract cache states with the join operator.*

Proof. If only pairs of disjuncts which satisfy this property are merged, then for any block $b \in \mathcal{B}$, if b is not aged in either disjunct, then it will not be aged in their abstract union. This corresponds to the case where there is no precision loss due to using the $\mathcal{C}^\sharp$ abstract domain compared to $\mathcal{P}(\mathcal{C}^\sharp)$. ◀

▶ **Corollary 14.** *If $t = n - 1$, then merging based on the aforementioned strategy will prevent a memory block from becoming possibly uncached as a result of merging two abstract states.*

Proof. This is just a special case of Lemma 12, where $t = n - 1$. ◀

The two corollaries could lead to two different merging strategies to serve as similarity relations in the definition of the abstract join operator. The first being to prevent the spurious aging of any memory block as a result of merging two abstract cache states, and the second being to prevent any block from becoming spuriously uncached in the same scenario. While the two merging strategies have properties that are desirable, they have their limitations in terms of their utility in practice. First, it may require many disjuncts to be able to merge only according to the strategy suggested by Corollary 13. Second, using the strategy suggested by Corollary 14, the number of disjuncts required may be large, but furthermore, the user is forced to pick a specific t $(n-1)$.

Therefore, in practice, we merge two states according to the following similarity relation: $\forall b \in \mathcal{B}. \max_{\neq n}(C(b)) = \max_{\neq n}(C'(b))$ (when considering more than one cache set ($Cset$), the formula becomes $\exists CSet . \forall b \in CSet. \max_{\neq n}(C(b)) = \max_{\neq n}(C'(b))$). That is, if there is (a cache set such that) no memory block whose maximum (non-associativity) age differs between C and C', then we merge the two abstract states.

The goal is to encourage falling into either of the two cases where Lemma 12 indicates that the merging will not lead to spurious aging, while still keeping the number of disjunctions manageable by enforcing less stringent requirements than suggested by either Corollary 13 or Corollary 14. Of course, many other relations could be used, including strategies based on the syntax and semantics of the program, which we intend to explore in future work.

4.2.5 The Widening Operator

Finally, the last domain operation to be defined is the widening operator. A widening operator serves to enforce termination of analyses which use abstract domains with infinitely increasing chains, or to speed up the analysis, regardless of the abstract domain. Given that $\mathcal{C}^\sharp$ has finite height, no widening is required for it. However, $\mathcal{V}^\sharp$ abstracts program values and therefore may not be of finite height; thus we must introduce a widening operator for $\mathcal{VC}^\sharp$.

One way to instantiate a widening operator for the finite powerset domain is the through the use of a *cardinality-based* widening [3], which is what we do in our instantiation of the framework. The formal details of the cardinality-based widening (written as slight adaptations from the details in [3]) are shown in Figure 8. In a nutshell, cardinality-based widening ensures termination by first ensuring that the cardinality of the *widening* argument is bounded by a fixed (user-specified) size k. Then, a reduction map, which takes a set of elements and removes and replaces pairs of elements where one strictly subsumes the other by the widening of these two elements, is applied in a recursive manner until the set no

Egli-Milner Partial Order $\sqsubseteq_{EM}$ ($\sqsubseteq_{EM}: \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \times \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \to \{\mathtt{tt}, \mathtt{ff}\}$). The Egli-Milner partial order is defined as follows: $S \sqsubseteq_{EM} S' \iff S = \emptyset \vee (\forall s \in S \,.\, \exists s' \in S' \,.\, s \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s' \wedge \forall s' \in S'.\exists s \in S.s \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s')$.

k-Collapsor ($\Uparrow_k: \mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \to \mathcal{VC}^\sharp$). Given $S \in \mathcal{VC}^\sharp$ such that S is non-redunandant according to $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$, a k-Collapsor $\Uparrow_k (S)$ yields $S' \in \mathcal{VC}^\sharp$ such that $S \sqsubseteq_{EM} S'$ and, moreover, $|S'| \leq k$.

∇-Reduction Map ($\Omega^\nabla : \mathcal{P}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \to \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$). The ∇-Reduction map is defined recursively: $\Omega^\nabla(S) := ite(\exists s, s' \in S \,.\, s \sqsubset_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s', \Omega^\nabla((S \setminus \{s, s'\}) \cup \{s \nabla_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s'\}), S)$.

Widen for $\mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ (${}_k\nabla_P : \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \times \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp) \to \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$). Given $S, S' \in \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ such that $S \sqsubset_{\mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)} S'$, $S_k \nabla_P S' := \Omega^\nabla(S \cup S'')$, where $S'' :=\Uparrow_k (S')$.

Widen for $\mathcal{VC}^\sharp$ (${}_k\nabla_P : \mathcal{VC}^\sharp \times \mathcal{VC}^\sharp \to \mathcal{VC}^\sharp$). Given $S, S' \in \mathcal{VC}^\sharp$ such that $S \sqsubset_{\mathcal{VC}^\sharp} S'$, $S_k \nabla_P S'' := \Omega^\nabla(S \cup S'')$, where $S'' := S' =\Uparrow_k (S')$, by virtue of $|S'| \leq k$, as it is a member of $\mathcal{VC}^\sharp$.

Figure 8 Details of the widening operator.

longer changes. Together, the two form a ∇-connected extrapolation heuristic [3], which lifts the base-level widening $\nabla_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$ to the powerset domain, while preventing unbounded growth, guaranteeing termination.

In the case of the finite powerset domain of non-redundant (w.r.t. $\sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}$) sets *without* a restriction on the cardinality sets (whose elements are denoted, with a slight abuse of notation, by $\mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$ in Figure 8), the two steps are accomplished by a using a k-Collapsor ($\Uparrow_k$) and the reduction map Ω^∇. The k-Collapsor takes an element of $\mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$, S, and returns an element $S' \in \mathcal{P}_{fn}(\mathcal{V}^\sharp \times \mathcal{C}^\sharp)$, such that $|S'| \leq k$ and $S \sqsubseteq_{EM} S'$. The Elgi-Milner partial order ($S \sqsubseteq_{EM} S'$) means that for two sets S, S', every element in S is overapproximated by an element in S' AND every element in S' overapproximates some element in S. There are many ways to define a k-Collapsor [3], but it is worth noting that our join operator defined in Algorithm 1, can be used to define a k-Collapsor, e.g., $S \mapsto S \oplus^*_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} S$. By construction, the resulting set is of size less than or equal to k. Furthermore, since elements of S are merged, every element in the resulting set subsumes some element in S, enforcing $S \sqsubseteq_{EM} S \oplus^*_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} S$.

In our abstract domain, $\mathcal{VC}^\sharp$, where each set is restricted to be of size at most k, the widening operator for $\mathcal{VC}^\sharp$ can be defined as shown at the bottom of Figure 8. That is, the k-Collapsor is the identity function, as all elements in the domain are bounded by cardinality k. The termination and soundness guarantees follow from the results established in [3].

5 Experiments

We have implemented our method in a static program analyzer designed for quantifying cache side-channel leakage. Our analyzer is written in OCaml and built upon the CacheAudit analysis framework, the state-of-the-art tool for computing upper bounds of cache side-channel leakage via abstract interpretation. Our techniques are implemented as functors for the existing CacheAudit abstract domains, transforming the existing abstract interpreter to gain precision in the key ways we identified.

5.1 Experimental Setup

We conducted all experiments on a computer with an Intel Xeon W-2245 CPU and 128 GB RAM, running Ubuntu 20.04 operating system. The experiments were designed to answer the following questions:

- **RQ1**. Do the upper bounds computed by our method improve upon the state-of-the-art?
- **RQ2**. Do the two innovative techniques presented in Section 4 have a synergistic effect in practice?

Our benchmark consists of 29 C programs that implement a variety of sorting algorithms and cryptographic functions. For every sorting algorithm, we introduce a "structured" version, meaning that the elements of the arrays to be sorted are data structure types, consisting of several other components: character arrays and integers. This set-up reflects real-world applications of algorithms that carry some "informational payload".

Each sorting algorithm was assumed to run on an array of size 24. While loop unrolling is not strictly necessary for abstract interpretation based methods, it was required for certain programs (independent of the technique used), and thus, we allowed a loop unrolling limit of 1024 for those programs, as recommended by the tool. We limit the maximum number of disjunctions to 10.

5.2 Results for Answering RQ1

To answer RQ1, i.e., do the upper bounds computed by our method improve upon the state-of-the-art, we compare the results of our method and the existing method on all 29 benchmark programs. The results are shown in Table 1. Column 1 shows the name of the benchmark program. Columns 2-4 compare leakage quantification results for two types of adversaries: SM stands for the shared-memory adversary and DM stands for the disjoint-memory adversary. In general, the leakage for DM is smaller than or equal to the leakage for SM. In both cases, the quantification results of the existing and new methods are measured in bits – a smaller number means a better result (less leakage). In Column 4, the ✔symbol means that our method obtains a better result, and the ✔symbol means that our method obtains the best-possible result (e.g., when the leakage is already 0). Columns 5-6 compare the total analysis time in seconds. In general, we find that in most cases the running time is about twice as long compared to the baseline methodology. This is expected due to the extra domain operations required due to refining the $\mathcal{C}^\sharp$ domain to use *sets* of abstract states.

Table 1 shows that the new method obtains either better or the best-possible quantification results on 13/15 benchmark programs that implement various sorting algorithms. For example, the quantification results for *cocktailsortstruct* and *shellsort* are the best-possible because the leakages obtained by our method are equal to 0. On the other 2/15 benchmark programs (*gnomesort* and *shellsortstruct*), the new method obtains quantification results that are as good as those of the existing method. As for the benchmark programs that implement cryptographic algorithms, the new method obtains either better quantification results on 9/14 of them, and obtains the same results as the existing method on 5/14 of them. We note that the results are also dependent on the cache configuration used. For example, using a $32K$ cache with associativity 16, and line size 32, on *sosemanuk* and *hc-128*, in particular, the precision of the leakage improves by close to 10 bits and 13 bits by using our method, corresponding to elimination of 1024 and 8192 spurious cache states, respectively. Overall, the results show that the upper bounds computed by our method improve upon the state-of-the-art significantly.

5.3 Results for Answering RQ2

To answer RQ2, i.e., do the two techniques presented in Section 4 have a synergistic effect in practice, we conducted an ablation study, by enabling each individual technique and comparing it against the state-of-the-art. These comparisons were conducted on all 29

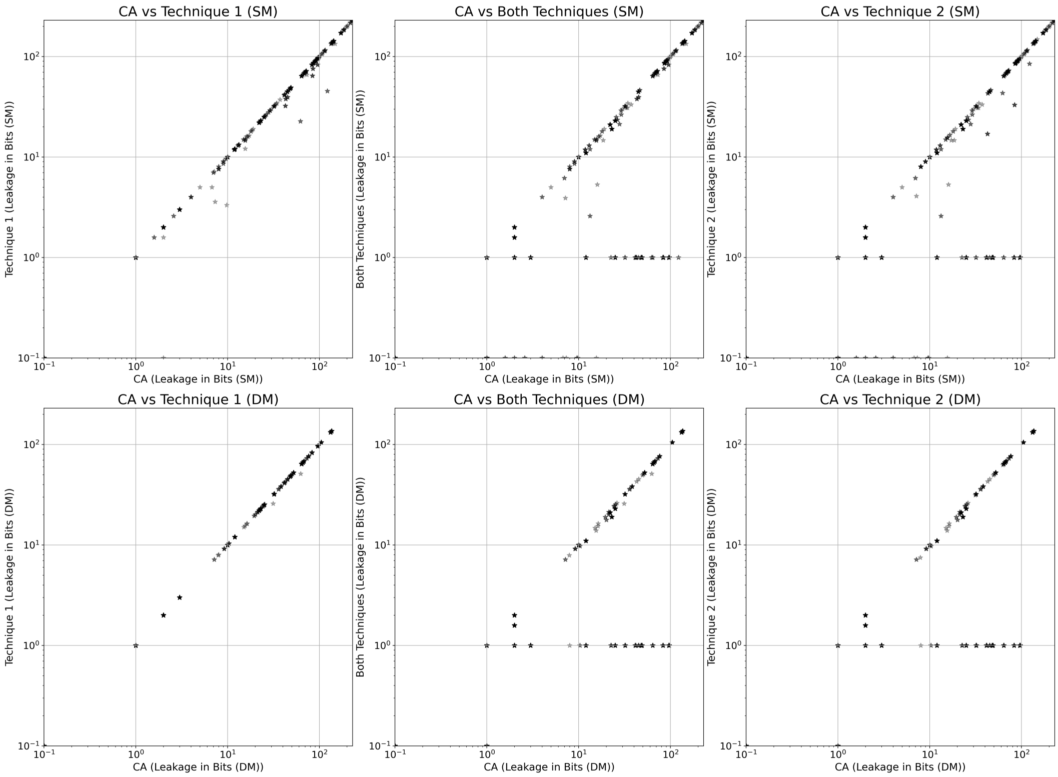

Figure 9 Evaluating the impact of the two new techniques in our method, by comparing them against the existing method. CA is the existing method (CacheAudit), Technique 1 is the first new technique in our method (the new abstract transfer function), Technique 2 is the second new technique in our method (refining the $\mathcal{C}^\sharp$ domain), and Both Techniques is our method with both of the two new techniques. The scatter plots on top are for the SM adversaries, while the scatter plots at the bottom are for the DM adversaries. In all of these scatter plots, points below the diagonal line ($y = x$) are winning cases for our method against the existing method.

benchmarks programs, with various cache settings. That is, we set the cache size S to 4KB, 8KB, 16KB, 32KB and 64KB, the associativity n to 4, 8, and 16, and the cache line size L to 32 and 64 bytes. While we recognize that 16 is not a common associativity for real-world caches, our goal was to stress-test our abstract transformer under higher associativities. The results are shown as scatter plots in Figure 9. In each scatter plot, the x-axis is the leakage (in bits) obtained by the state-of-the-art method, and the y-axis is the leakage (in bits) obtained by our method (with one or both techniques enabled). Thus, the diagonal line represents the cases where our method is tied with the existing method, whereas points below the diagonal line are winning cases for our method.

In Figure 9, the two scatter plots on the left-hand side show the effectiveness of the new abstract transfer function. While most of the points are on the diagonal line, meaning that the two methods are tied, there are some points that are significantly below the diagonal line, indicating the effectiveness of the proposed technique for these cases. The two scatter plots on the right-hand side show the effectiveness of using disjunctions. Many of the points are below the diagonal line, which are the winning cases for our method. The two scatter plots in the middle show that using both the new abstract transfer function and leveraging disjunctions together perform very well, with even more points below the diagonal line.

We also collected detailed results of the experimental comparison, which are shown in Table 2. Various cache settings were used in the experiments, as shown in Column 2. For brevity, we only show these detailed results for a representative benchmark program named *scatter_gather*.

Overall, the results show that each of the two techniques is effective in isolation; furthermore, when the two techniques are used together, they often have a synergistic effect in terms of improving the precision of leakage quantification.

6 Related Work

As mentioned earlier, the most closely related work is that of Doychev et al. [11], which we regard as the baseline algorithm for quantifying cache side-channel leakage. The key difference in our work is a new abstract transfer function and a disjunctive refinement for increasing the precision of abstract interpretation.

Doychev et al. [11] support other adversaries, including trace-based and timing adversaries. Given that our abstractions fundamentally improve the precision of the abstract cache states in an abstract domain that is specialized for quantification, we expect that our techniques will help improve quantification results on downstream static analyses that rely on abstract cache states.

Kopf et al. [17] target cache-based adversaries, and conduct quantification via counting formulae, combined with an abstract interpreted-based static analysis. They recognize that trace partitioning [23] during the static analysis led to increased precision in the quantification results. However, trace-partitioning was conducted by manual program transformation, whereas our method is automated via abstract interpretation to parsimoniously leverage disjunctive information. Beyond abstract interpretation, which is a *sound* analysis technique, there are methods based on alternative analysis techniques such as bounded modeling checking [20] or symbolic execution [7]. However, their results may not be sound.

There are also methods targeting other kinds of adversaries. In the case of trace-based adversaries, it is assumed that a malicious attacker may observe the sequences of memory accesses throughout program execution; thus, quantification techniques aim to compute an upper bound on the number of distinct memory access traces possible. Various tools have been developed to compute an upper bound for the number of possible distinct memory access traces. Ma et al. [20] introduce an abstraction known as differential set that tracks, for each memory access, all possible addresses that might be accessed by that operation or its “sibling” operations in other control flows. Their abstraction is combined with bounded model-counting to compute a sound upper bound of information leakage. Other works leverage techniques such as symbolic execution to compute these upper bounds.

Beyond quantification, there are methods for cache hit/miss classification [26, 25, 14]. In particular, Touzeau et al. [25] combine abstract interpretation with model checking to classify memory accesses in LRU caches as “always hit”, “always miss”, or “definitely unknown”. Touzeau et al. [26] also introduce a method that represents cache states using anti-chains of minimal/maximal elements rather than full state sets, thus enabling efficient computation while preserving precision. Gysi et al. [14] introduce symbolic techniques to count cache misses without having to enumerate all memory accesses, making the analysis practical through a hybrid approach that combines symbolic computation with selective enumeration. However, these works are not designed for quantifying cache side-channel leakage.

7 Conclusion

We have presented a method for significantly improving the precision of abstract interpretation based static analysis for quantifying cache side-channel leakage. The method uses a new abstract transfer function to prevent spurious aging and abstract domain refinement during the analysis step, which uses disjunctions parsimoniously to prevent spurious combinations of cache states. Our experimental evaluation on benchmark programs consisting of sorting and cryptographic algorithms shows that the method is more accurate in quantifying cache side-channel leakage than the state-of-the-art technique. Furthermore, both of the two new techniques in our method contribute to the performance improvement.

Table 1 Comparing existing methods (**B** [11]) and our new method (**NM**) on a 32KB cache with associativity 8 and line size 32.

Program	Leakage Quantification			Time (s)	
	B (SM / DM)	**NM** (SM / DM)	Comp.	**B**	**NM**
bingosort	1.0 / 1.0	0.0 / 0.0	✔	4	18
bingosortstruct	25.0 / 25.0	23.0 / 23.0	✔	101	227
bubblesort_opt	3.0 / 3.0	1.0 / 1.0	✔	0	1
bubblesort_opt_struct	25.0 / 25.0	1.0 / 1.0	✔	2	8
bubblesort_struct	1.0 / 1.0	0.0 / 0.0	✔	2	5
cocktailsort	0.0 / 0.0	0.0 / 0.0	✔	17	45
cocktailsortstruct	1.0 / 1.0	0.0 / 0.0	✔	108	284
gnomesort	2.0 / 2.0	2.0 / 2.0	same	3	9
gnomesortstruct	23.0 / 23.0	19.0 / 19.0	✔	38	75
iterativeheapify	2.0 / 2.0	1.0 / 1.0	✔	11	21
iterativeheapifystruct	22.0 / 22.0	21.0 / 21.0	✔	92	146
odd_even_sort	0.0 / 0.0	0.0 / 0.0	✔	8	20
odd_even_sort_struct	1.0 / 1.0	0.0 / 0.0	✔	54	148
shellsort	0.0 / 0.0	0.0 / 0.0	✔	0	1
shellsortstruct	1.0 / 1.0	1.0 / 1.0	same	1	4
defensive_gather	96.0 / 96.0	1.0 / 1.0	✔	14	38
scatter_gather_openssl_1_0_2	97.0 / 97.0	1.0 / 1.0	✔	2	3
window_mod_exp_libgcrypt_161	2.0 / 2.0	1.5 / 1.5	✔	1	1
window_mod_exp_libgcrypt_163	0.0 / 0.0	0.0 / 0.0	✔	1	1
rabbit	0.0 / 0.0	0.0 / 0.0	✔	4	14
salsa	0.0 / 0.0	0.0 / 0.0	✔	4	10
aes-128-preloading	15.6 / 0.0	14.5 / 0.0	✔	14	51
aes-192-preloading	15.6 / 0.0	15.0 / 0.0	✔	16	82
aes-256-preloading	16.5 / 0.0	16.0 / 0.0	✔	23	123
aes-128-rom	142.5 / 132.7	141.5 / 132.6	✔	23	63
aes-192-rom	142.6 / 132.6	142.1 / 132.6	✔	26	92
aes-256-rom	143.2 / 132.6	142.6 / 132.6	✔	34	114
sosemanuk	64.0 / 64.0	64.0 / 64.0	same	90	191
hc-128	29.0 / 0.0	29.0 / 0.0	same	2242	3853

Table 2 Evaluating the impact of the two new techniques in our method on the benchmark program *scatter_gather_openssl_1_0_2* using various cache settings. S is the cache size in bytes, n is the associativity level, and L is the cache line size in bytes.

Cache Setting	Leakage Quantification			Time (s)		
(S, n, L)	Technique-1 (SM / DM)	Ours (both) (SM / DM)	Technique-2 (SM / DM)	Tech-1	Ours (both)	Tech-2
(4096, 4, 32)	64.3 / 64.3	1.0 / 1.0	33.0 / 1.0	5	6	3
(4096, 4, 64)	32.3 / 32.3	1.0 / 1.0	17.0 / 1.0	3	3	3
(4096, 8, 32)	45.2 / 45.1	1.0 / 1.0	84.7 / 1.0	8	9	3
(4096, 8, 64)	22.6 / 22.6	1.0 / 1.0	43.3 / 1.0	5	5	4
(8192, 4, 32)	83.3 / 83.3	1.0 / 1.0	1.0 / 1.0	3	4	2
(8192, 4, 64)	41.9 / 41.9	1.0 / 1.0	1.0 / 1.0	2	2	2
(8192, 8, 32)	64.3 / 64.3	1.0 / 1.0	33.0 / 1.0	5	5	2
(8192, 8, 64)	32.3 / 32.3	1.0 / 1.0	17.0 / 1.0	3	3	3
(16384, 4, 32)	97.0 / 97.0	1.0 / 1.0	1.0 / 1.0	3	4	3
(16384, 4, 64)	49.0 / 49.0	1.0 / 1.0	1.0 / 1.0	2	2	3
(16384, 8, 32)	83.3 / 83.3	1.0 / 1.0	1.0 / 1.0	3	4	2
(16384, 8, 64)	41.9 / 41.9	1.0 / 1.0	1.0 / 1.0	2	2	3
(32768, 4, 32)	97.0 / 97.0	1.0 / 1.0	1.0 / 1.0	4	6	2
(32768, 4, 64)	49.0 / 49.0	1.0 / 1.0	1.0 / 1.0	2	4	1
(32768, 8, 32)	97.0 / 97.0	1.0 / 1.0	1.0 / 1.0	3	3	3
(32768, 8, 64)	49.0 / 49.0	1.0 / 1.0	1.0 / 1.0	2	3	2
(64512, 4, 32)	97.0 / 97.0	1.0 / 1.0	1.0 / 1.0	5	5	2
(64512, 4, 64)	49.0 / 49.0	1.0 / 1.0	1.0 / 1.0	3	3	3
(64512, 8, 32)	97.0 / 97.0	1.0 / 1.0	1.0 / 1.0	4	5	2
(64512, 8, 64)	49.0 / 49.0	1.0 / 1.0	1.0 / 1.0	2	4	1

References

1 Onur Aciiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, pages 80–91. IEEE Computer Society, 2007. doi:10.1109/FDTC.2007.4318988.

2 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 53–70. USENIX Association, 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.

3 Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening operators for powerset domains. *Int. J. Softw. Tools Technol. Transf.*, 9(3-4):413–414, 2007. URL: https://doi.org/10.1007/s10009-007-0029-y, doi:10.1007/S10009-007-0029-Y.

4 Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. *J. Comput. Secur.*, 27(1):137–163, 2019. doi:10.3233/JCS-181136.

5 Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In William Enck and Collin Mulliner, editors, *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX Association, 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

6 Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*, pages 4:1–4:6. ACM, 2017. doi:10.1145/3152701.3152706.

7 Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. Quantifying the information leakage in cache attacks via symbolic execution. *ACM Trans. Embed. Comput. Syst.*, 18(1):7:1–7:27, 2019. doi:10.1145/3288758.

8 Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy (SP 2009), 17-20 May 2009, Oakland, California, USA*, pages 45–60. IEEE Computer Society, 2009. doi:10.1109/SP.2009.19.

9 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

10 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979. doi:10.1145/567752.567778.

11 Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 431–446. USENIX Association, 2013. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev.

12 Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 406–421. ACM, 2017. doi:10.1145/3062341.3062388.

13 David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy, SP 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505. IEEE Computer Society, 2011. doi:10.1109/SP.2011.22.

14 Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. A fast analytical model of fully associative caches. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 816–829. ACM, 2019. doi:10.1145/3314221.3314606.

15 Chao Wang Jacqueline Mitchell. Quantifying cache side-channel leakage by refining set-based abstractions (extended version). https://github.com/jlmitche23/ecoop25CacheQuantification, 2025.

16 Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. doi:10.1007/3-540-68697-5_9.

17 Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 564–580. Springer, 2012. doi:10.1007/978-3-642-31424-7_40.

18 Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. Cross-origin pixel stealing: timing attacks using CSS filters. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1055–1062. ACM, 2013. doi:10.1145/2508859.2516712.

19 Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 605–622. IEEE Computer Society, 2015. doi:10.1109/SP.2015.43.

20 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):1470–1498, 2023. doi:10.1145/3622850.

21 Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. doi:10.1007/11605805_1.

22 Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212. ACM, 2009. doi:10.1145/1653662.1653687.

23 Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5):26, 2007. doi:10.1145/1275497.1275501.

24 Isabell Schmitt and Sebastian Schinzel. Waffle: Fingerprinting filter rules of web application firewalls. In Elie Bursztein and Thomas Dullien, editors, *6th USENIX Workshop on Offensive Technologies, WOOT'12, August 6-7, 2012, Bellevue, WA, USA, Proceedings*, pages 34–40. USENIX Association, 2012. URL: http://www.usenix.org/conference/woot12/waffle-fingerprinting-filter-rules-web-application-firewalls.

25 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Ascertaining uncertainty for efficient exact cache analysis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2017. doi:10.1007/978-3-319-63390-9_2.

26 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Fast and exact analysis for LRU caches. *Proc. ACM Program. Lang.*, 3(POPL):54:1–54:29, 2019. doi:10.1145/3290367.

27 Tom van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1382–1393. ACM, 2015. doi:10.1145/2810103.2813632.

28 Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 859–874. ACM, 2017. doi:10.1145/3133956.3134016.

29 Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time RSA. *J. Cryptogr. Eng.*, 7(2):99–112, 2017. URL: https://doi.org/10.1007/s13389-017-0152-y, doi:10.1007/S13389-017-0152-Y.

30 Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 305–316. ACM, 2012. doi:10.1145/2382196.2382230.

A 14 Concrete Cache States

Concrete Cache States
$c_1 = \{a \mapsto 1, b \mapsto 4, c \mapsto 4, d \mapsto 0\}$
$c_2 = \{a \mapsto 1, b \mapsto 2, c \mapsto 4, d \mapsto 0\}$
$c_3 = \{a \mapsto 1, b \mapsto 4, c \mapsto 2, d \mapsto 0\}$
$c_4 = \{a \mapsto 4, b \mapsto 1, c \mapsto 4, d \mapsto 0\}$
$c_5 = \{a \mapsto 3, b \mapsto 1, c \mapsto 2, d \mapsto 0\}$
$c_6 = \{a \mapsto 3, b \mapsto 2, c \mapsto 1, d \mapsto 0\}$
$c_7 = \{a \mapsto 1, b \mapsto 2, c \mapsto 3, d \mapsto 0\}$
$c_8 = \{a \mapsto 4, b \mapsto 1, c \mapsto 2, d \mapsto 0\}$
$c_9 = \{a \mapsto 2, b \mapsto 4, c \mapsto 1, d \mapsto 0\}$
$c_{10} = \{a \mapsto 4, b \mapsto 4, c \mapsto 1, d \mapsto 0\}$
$c_{11} = \{a \mapsto 2, b \mapsto 1, c \mapsto 4, d \mapsto 0\}$
$c_{12} = \{a \mapsto 4, b \mapsto 4, c \mapsto 4, d \mapsto 0\}$
$c_{13} = \{a \mapsto 2, b \mapsto 3, c \mapsto 1, d \mapsto 0\}$
$c_{14} = \{a \mapsto 2, b \mapsto 1, c \mapsto 3, d \mapsto 0\}$

B Proof that $Upd^\sharp_*$ is Sound: $\alpha_{\mathcal{C}^\sharp} \cdot Upd_w \cdot \gamma_{\mathcal{C}^\sharp}(C) \sqsubseteq_{\mathcal{C}^\sharp} Upd^\sharp_*(C, w)$

Proof. To show this, we will prove that for every memory block b, if there is some concrete state $c' \in Upd_w \cdot \gamma_{\mathcal{C}^\sharp}(C)$, such that $c'(b) = a$, then $a' \in Upd^\sharp_*(C, w)(b)$. For brevity, let $C' = Upd^\sharp_*(C, w)$.

We will proceed in cases:

1. **Case** $(b = w)$. This is immediate. Then for all $c' \in Upd_w \cdot \gamma_{\mathcal{C}^\sharp}(C)$, $c'(b) = 0$. By the fourth case in $Upd^\sharp_*$'s definition $0 \in C'(b)$.
2. **Case** $(b \neq w)$. We will now consider $c' = Upd(c, w)$. Let $c(b) = a$ and $c'(b) = a'$.
   a. **Subcase: $(a = a' = n)$**. If this is the case, then it immediately follows, due to the definition of Set O in $Upd^\sharp_*$. Hence $a' = n \in C'(b)$.
   b. **Revised Subcase: $(a = a' < n)$**. In this case, we will show that if there is some $c \in \gamma_{\mathcal{C}^\sharp}(C)$ such that $c'(b) = a$, then $a \in C'(b)$. $c'(b) = a$ means that $c(w) < c(b)$. Given that C overapproximates the concrete states, $c(w) \in C(w)$ and $c(b) \in C(b)$. This means that there is some c_w, c_b such that $c_w < c_b$. Therefore, by Case $O_>\langle w \rangle$ in $Upd^\sharp_*$, $c_b = a \in C'(b)$.
   c. **Revised Subcase:** $(a' = a + 1)$. In this case, we show that if there is some $c \in \gamma_{\mathcal{C}^\sharp}(C)$, such that $c'(b) = a + 1$, then $a + 1 \in C'(b)$. Consider when $Upd^\sharp_*$ increments $c_b (= a)$. It occurs when the following condition is true: $c_b > c_w \wedge \neg(w \in Var(a) \wedge |Var(C, a) \backslash \{b\}| \leq a)$. If $c'(b) = a + 1 = c_b + 1$, then it must be the case that there is some c_w such that that $c(w) = c_w$ and $c_w > c_b$. Therefore, the first part of the condition holds. Now, suppose for the sake of contradiction that the second part of the condition does *NOT* hold. This means that $w \in Var(a) \wedge |Var(C, a) \backslash \{b\}| \leq a$ holds (note: it is the negated condition). If this holds, then because C overapproximates the concrete states, then $|Var(C, a) \backslash \{b\}| \leq a$ means that there can be at most a memory blocks (excluding b) which can be younger than a (e.g. when b is of age a), with w being one of them. In an LRU cache, if b is of age a, then there must be memory blocks occupying all ages $a - 1$ to 0. By the condition, w must be one of these blocks and there at most a such

blocks. Thus, because C overapproximates all concrete states, this would imply that there is no concrete state c such that $c(b) = a = c_b$ and $c_w > c_b$. This is contradiction with our original assumption that $c(w) = c_w$ and $c_w > c_b$. Thus, our claim is proved.

Through all the enumerated cases, we have shown that if $c'(b) = a'$, then $a' \in C'(b)$. ◀

C Validity of the Meet and Join Operators of $\mathcal{VC}^\sharp$

C.1 Validity of the Meet Operator

Here, we will show that the meet operator $S \sqcap_{\mathcal{VC}^\sharp} S'$, defined in Section 4.2.2 as $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(\{s \sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s' \mid s \in S, s' \in S'\})$ is sound. Specifically, for any $S, S' \in \mathcal{VC}^\sharp$, we show that $\gamma(S) \cap \gamma(S') \subseteq \gamma(S \sqcap_{\mathcal{VC}^\sharp} S')$.

Proof. Let $(v, c) \in \gamma(S) \cap \gamma(S')$. Then, $(v, c) \in \gamma(S)$ and $(v, c) \in \gamma(S')$. Given that $(v, c) \in \gamma(S)$, there exists $s \in S$ such that $(v, c) \in \gamma_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(s)$. Similarly, there exists s' such that $(v, c) \in \gamma_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(s')$. Therefore $(v, c) \in \gamma_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(s \sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s')$. By the definition of $\sqcap_{\mathcal{VC}^\sharp}$, we know that some abstract value in $S \sqcap_{\mathcal{VC}^\sharp} S'$ must overapproximate $s \sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s'$. Thus, $(v, c) \in \gamma(S \sqcap_{\mathcal{VC}^\sharp} S')$. Hence, we have proved the statement. ◀

We note that if $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(\{s \sqcap_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s' \mid s \in S, s' \in S'\})$ replaces line 1 in Algorithm 1, that the result is still sound. The proof is similar to proof of soundness for the join operator Appendix C.2, so we do not include it here.

C.2 Validity of the Join Operator

Here, we will show that the join operator $\oplus^*_{\mathcal{VC}^\sharp}$, defined in Algorithm 1 is sound. Specifically, we will show that for any $S, S' \in \mathcal{VC}^\sharp$ that $\gamma(S) \cup \gamma(S') \subseteq \gamma(S \oplus^*_{\mathcal{VC}^\sharp} S')$.

Proof. Let $(v, c) \in \gamma(S) \cup \gamma(S')$. Then, without loss of generality, assume that $(v, c) \in \gamma(S)$. Thus, there is some $s \in S$ such that $(v, c) \in \gamma_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(s)$. We know that there is some abstract state which overapproximates s in $\Omega_R^{\mathcal{V}^\sharp \times \mathcal{C}^\sharp}(S \cup S')$. Furthermore, Algorithm 1 only merges abstract states or removes redundant ones, so there remains some value s'' in the resulting join such that $s \sqsubseteq_{\mathcal{V}^\sharp \times \mathcal{C}^\sharp} s''$. Therefore, $(v, c) \in \gamma(S \oplus^*_{\mathcal{VC}^\sharp} S')$. ◀

D Proof of Lemma 12.

Proof. For the purposes of the proof, let $(\star)$ denote the condition: $\forall b \in \mathcal{B}. \max_{<t}(C(b)) = \max_{<t}(C'(b))$. If $Upd^\sharp_*(C, w)$ does not age b from t to $t+1$, then $\max c_w < t$ or $|Var(C, t) \backslash \{b\}| \leq t \wedge w \in Var(C, t) \backslash \{b\}$. Similarly, if $Upd^\sharp_*(C', w)$ does not age b from t to $t+1$, then this means that $\max c'_w < t$ or that $|Var(C', t) \backslash \{b\}| \leq t \wedge w \in Var(C', t) \backslash \{b\}$. In the following, let $C'' = C \sqcup_{\mathcal{C}^\sharp} C'$.

We will proceed by cases:

(1) If $\max c_w < t$ and $\max c'_w < t$. This indicates that $\max c''_w < t$, therefore $Upd^\sharp(C'', w)$ will not age b from t to $t+1$.

(2) Consider WLOG that C satisfies $\max c_w < t$ and that C' satisfies $|Var(C', t) \backslash \{b\}| \leq t \wedge w \in Var(C', t) \backslash \{b\}$. Furthermore, we know that if $w \in Var(C', t) \setminus \{b\}$, then $w \in Var(C'', t) \setminus \{b\}$. Thus, the only way that $Upd^\sharp_*(C'', w)$ will age b from t to $t+1$ is if there exists block v such that $\exists c_v \in C(v). c_v < t$ and $\min C'(v) \geq t$, such that $|Var(C'', t) \setminus \{b\}| > t$. This implies that $\max_{<t} C(v)$ is some value larger than $-\infty$ and $\max_{<t} C'(v) = -\infty$. This is a contradiction with $(\star)$.

(3) Lastly, consider the case where C and C' satisfy $|Var(C,t)\backslash\{b\}| \leq t \wedge w \in Var(C,t)\backslash\{b\}$, and $|Var(C',t)\backslash\{b\}| \leq t \wedge w \in Var(C',t)\backslash\{b\}$, respectively. If $w \in Var(C,t) \setminus \{b\}$ or $w \in Var(C',t)\setminus\{b\}$, then $w \in Var(C'',t)\setminus\{b\}$. Due to $(\star)$, $Var(C,t)\setminus\{b\} = Var(C',t)\setminus\{b\}$. Suppose not. Then this implies WLOG, that is some memory block v such that $v \in Var(C,t) \setminus \{b\}$ and $v \notin Var(C',t) \setminus \{b\}$. This means that $\min C'(v) \geq t$. Hence, this means that $\max_{<t} C'(v) = -\infty$ and $\max_{<t} C(v) \neq -\infty$, a contradiction with $(\star)$.

Hence, by all three cases, we have proved our claim. ◀