Abstract Interpretation under Speculative Execution

Meng Wu
Virginia Tech
Blacksburg, VA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

Abstract

Analyzing the behavior of a program running on a processor that supports speculative execution is crucial for applications such as execution time estimation and side channel detection. Unfortunately, existing static analysis techniques based on abstract interpretation do not model speculative execution since they focus on functional properties of a program while speculative execution does not change the functionality. To fill the gap, we propose a method to make abstract interpretation sound under speculative execution. There are two contributions. First, we introduce the notion of *virtual control flow* to augment instructions that may be speculatively executed and thus affect subsequent instructions. Second, to make the analysis efficient, we propose optimizations to handle merges and loops and to safely bound the speculative execution depth. We have implemented and evaluated the proposed method in a static cache analysis for execution time estimation and side channel detection. Our experiments show that the new method, while guaranteed to be sound under speculative execution, outperforms state-of-the-art abstract interpretation techniques that may be unsound.

CCS Concepts • **Software and its engineering → Formal software verification**; **Compilers**; • **Security and privacy → Cryptanalysis and other attacks**.

Keywords Static analysis, speculative execution, abstract interpretation, timing side channel, WCET, cache

ACM Reference Format:
Meng Wu and Chao Wang. 2019. Abstract Interpretation under Speculative Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314647

1 Introduction

Speculative execution [53] is a feature that has been implemented by many modern processors. It allows a processor to increase the execution speed by exploring certain program paths ahead of time instead of waiting for the path conditions to be satisfied. This is to prevent slower instructions, e.g., memory accesses, from blocking faster instructions. For example, when a program reaches a branching instruction, e.g., `if(x>5){...}else{...}` where the condition depends on an uncached value of `x` stored in memory, a *non-speculative* execution will force the processor to wait, often for tens or hundreds of clock cycles, until `x` is loaded from memory, whereas *speculative* execution allows the processor to make a prediction of the branching target and then proceed to execute the predicted branch. During speculative execution, the processor maintains a checkpoint of the CPU's register state, which will be used to roll back the changes if the prediction turns out to be incorrect, i.e., after the value of `x` is fetched from memory. However, if the prediction turns out to be correct, speculative execution will save time and thus outperform non-speculative execution.

Speculative execution is designed to be *transparent* to the program running on the processor; that is, it does not affect the program semantics, as the rollback ensures that functional properties are preserved. This is the reason why, in the past, static analysis techniques do not model speculative execution. However, recent vulnerabilities such as Meltdown [36], Spectre [28] and ForeShadow [55] force the community to take another look because, although speculative execution preserves the CPU's register state, for performance reasons, it does not preserve the states of many other components such as the cache and the pipeline [22, 23].

To see why this may be a problem, consider the cache state that may be altered by speculative execution and thus affect the timing behavior of the subsequent non-speculative execution, e.g., cache hits may become misses, or vice versa. This is important because an instruction may take only 1-3 clock cycles when there is a cache hit, but tens or even hundreds of clock cycles when there is a cache miss. Static analysis is useful in examining the cache related properties of a program, e.g., to detect information leaks through timing side channels [7, 16, 29, 51, 58] or prove that a computation task always meets the deadline [18, 20].

For side channel detection, in particular, one may want to know if the program's execution time depends on secret data, e.g., the cryptographic key, security token, or password. For deadline estimation, one may want to know the maximum number of cache misses along program paths, since it corresponds to the execution time in the worst case. In both applications, static analysis must be *sound* to be useful. By sound, we mean all possible behaviors must be considered. The reason is because, if the analysis fails to take into consideration a certain behavior, e.g., a specific execution, it may

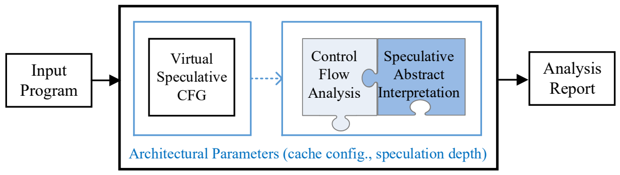
Input Program
Virtual Speculative CFG
Control Flow Analysis
Speculative Abstract Interpretation
Analysis Report
Architectural Parameters (cache config., speculation depth)

Figure 1. Our static program analysis framework based on sound abstract interpretation of speculative executions.

miss a bug or security vulnerability, which is not acceptable in critical applications.

Unfortunately, existing abstract interpretation techniques [16, 18, 19, 48] are unsound under speculative execution. Instead, these prior works on abstract interpretation focus more on modeling *non-speculative* executions, for which numerous techniques have been developed, including widening/narrowing, chaotic iteration, and efficient implementations of abstract domains. Under *speculative* execution, however, none of these techniques is relevant because the problem is no longer about removing infeasible paths from the over-approximated analysis, but about preventing real behaviors from being excluded. This requires a different set of ideas from what already exist in the literature.

We propose a method for lifting abstract interpretation algorithms so that they are sound again under speculative execution. We have developed two techniques. The first one is a unified way of modeling both non-speculative and speculative executions using *virtual control flow*. The second one is redundancy removal, which is crucial for reducing runtime overhead while maintaining accuracy.

At a high level, the virtual control flow augments the program's CFG by adding new nodes and edges, e.g., transitions from locations in the body of one speculatively executed branch to the starting point of the other branch, to model the rollback upon mis-prediction. This approach is generally applicable, regardless of how the abstract state is defined and which algorithm is used to compute the fixed point. For example, the abstract state may model side effects on the cache or pipeline [46, 47], the non-functional properties to be verified may be timing or power [17, 60], and the abstract domain may be interval [14] or octagonal [40].

We have implemented the method in a static cache analysis shown in Figure 1, to compute the memory accesses that correspond to *Must-Hits*. In this context, our method first computes all possible speculative paths of the program and uses them to augment the CFG. Next, it traverses the speculative CFG to perform abstract interpretation, which computes an abstract (cache) state for each program location. To reduce the runtime overhead, it also bounds the depth and number of speculative executions that abstract interpretation has to consider. This is possible because, in many cases, the abstract states are already computed for some location and thus can be used to bound the speculative executions in other locations. Furthermore, we discover that the accuracy of abstract interpretation is often affected by *when* abstract states from speculative and non-speculative executions are merged, and we develop a strategy named "just-in-time merging" to minimize the loss of accuracy.

We have implemented our method in LLVM [32], where the speculative CFG is constructed by an LLVM pass before it is used by abstract interpretation. We evaluated it on two types of benchmarks: cryptographic software and real-time software, where the goal is to detect timing side channels and to estimate the execution time, respectively. In both cases, the instruction set architecture is Alpha 21264, with 32-KB fully-associative data cache, 64 bytes per line, and the LRU replacement policy. Our experiments show that, compared to existing non-speculative methods, our method is able to detect significantly more timing related behaviors, i.e., cache misses and side-channel leaks. Furthermore, our optimizations are effective in reducing the runtime overhead while maintaining the accuracy.

To sum up, this paper makes the following contributions:

- We show why existing abstract interpretation techniques are unsound for speculative execution.
- We propose a method for lifting existing algorithms to make them sound for speculative execution.
- We develop optimizations to safely reduce the runtime overhead while maintaining the accuracy.
- We implement the method and demonstrate its effectiveness on a set of C programs.

The paper is structured as follows. First, we illustrate the problem and our solution in Section 2. Then, we provide the technical background in Section 3, before presenting our algorithms in Sections 4, 5 and 6. We present our experimental results in Section 7. We review the related work in Section 8. Finally, we give our conclusions in Section 9.

2 Motivation

We illustrate some scenarios in which speculative execution affects the cache behaviors associated with a program, and explain why such behaviors are crucial for execution time estimation and side channel detection.

2.1 Execution Time Estimation

Figure 2 shows a program that illustrates divergent cache behaviors under normal and speculative executions as observed in practice [2–4, 12, 25, 26]. Here, we have four variables: *ph*, *l1*, *l2*, and *p*, which are mapped to different cache lines. Suppose the register value *k* is 0, the load at line 8 will access *ph[0]*. We assume the cache has 512 lines in total and 64 bytes per line. We also assume the cache is fully associative, meaning any variable may be mapped to a different line. The place holder variable *ph* is mapped to the first 510 lines (line 3); in practice, *ph* may correspond to an assorted set of program variables. Each of the remaining variables, *l1*, *l2* and *p*, may be mapped to a cache line.

Depending on the branching condition, either *l1* or *l2* may be loaded to the cache, but both will result in 512 cache

```
1  char ph[64*510], l1[64], l2[64], p;
2  reg char k;
3  for(reg int i=0;i<64*510; i+=64) load ph[i];
4  if(p==0)
5   load l1[0];
6  else
7   load l2[0];
8  load ph[k];
```

Figure 2. Example program for timing side channel.

misses. As shown on the left-hand side of Figure 3, the statement at line 8, accessing *ph[0]*, is always a hit because the content is already in the cache.

However, under speculative execution, upon reaching the *if-else* statement, the CPU needs to load *p* from memory. Due to a cache miss, it performs a speculative execution of the branch `(p==0)`. If the branch prediction is incorrect and the CPU has to roll back the speculative execution, there will be 514 cache misses (among which 513 cache misses are observable from outside of the CPU) as shown by the right-hand-side trace in Figure 3.

In this case, the program first speculatively executes the *then*-branch and loads `l1` into the cache, and then rolls back to take the *else*-branch and loads `l2`. Although the functional side-effects of executing the *then*-branch are eliminated by the rollback mechanism, `l1` is already in the cache. Since the cache has only 512 lines, following the LRU replacement policy, the first line associated with *ph[0]* is evicted. This is why the subsequent access to *ph[0]* will be a cache miss.

For execution time estimation, the non-speculative execution will lead to 512 cache misses plus 1 cache hit, whereas the speculative execution will lead to 513 observable cache misses (and a speculative cache miss masked by the pipeline). The additional cache miss is important because it will cause a significant delay in the execution time. The message from this example is as follows: if a static analysis is not sound in modeling speculative execution, it may underestimate the worst-case execution time and produce a bogus proof that the computation task meets its deadline.

2.2 Side Channel Detection

We use Figure 2 again to illustrate a timing side channel made possible by speculative execution. That is, the attacker, by measuring the execution time of a program, may deduce information of the secret data. This time, we assume the variable k stores the secret data, e.g., a cryptographic key, and the value of k is used as an index to access an *S-Box*-like array named *ph*. If the time taken by the access varies with respect to k, there is an information leak.

In a non-speculative execution, there cannot be leaks in Figure 2 because, for all paths and values of k, the number of cache misses remains the same. In particular, accessing *ph[k]* is leak-free because the array is loaded to cache at line 3, and executing either branch at lines 5 and 7 will not evict it. However, similar to what we have observed in the execution time estimation example, speculative execution may execute one of the two branches first, and then roll back to execute the other branch. Since the memory locations associated with both branches must be accessed, which add up to more than 512 cache lines, some of the cache lines associated with *ph* will be evicted. Therefore, the subsequent *load* (at line 8) may be a cache miss. The difference in execution time may be observed by the attacker and used to deduce information of the secret k: whether the last statement leads to a cache miss depends on the value of k.

2.3 Technical Challenges

The above two examples illustrate the need to soundly model speculative execution. However, there are several challenges. The first one is to model the cache state of a program during speculative execution without drastically altering the abstract interpretation algorithm. The second challenge is to judiciously merge abstract states computed from normal and speculative executions, since *when* and *how* to merge them drastically affect the accuracy of the fixed-point computation. Furthermore, since a speculative execution may be rolled back at any moment, the number of scenarios is exponential in the number of speculatively executed instructions. If we have to enumerate, the analysis time will be prohibitively long. Therefore, we group scenarios into equivalence classes, based on which we perform reduction to balance the performance and accuracy.

In the remainder of this paper, we will present our solutions in detail.

3 Preliminaries

We review the basics of abstract interpretation, as well as the cache, branch prediction, and speculative execution.

3.1 Abstract Interpretation

Abstract interpretation [14] is a static analysis framework that considers all paths and inputs to obtain a sound over-approximation of the state at every program location [30, 31, 50]. For efficiency reasons, the state is kept *abstract* and often represented by a set of constraints in a certain *abstract domain*. For example, in the interval domain, each constraint is of the form $lb \leq x \leq ub$, where x is a variable and lb,ub are the lower and upper bounds. The join of two states, $s_1 = lb_1 \leq x \leq ub_1$ and $s_2 = lb_2 \leq x \leq ub_2$, is defined as $s_1 \sqcup s_2 = min(lb_1, lb_2) \leq x \leq max(ub_1, ub_2)$. Here, $\sqcup$ denotes the *join* operator, which returns an over-approximation of the set union. If, for example, the polyhedral abstract domain is used, a constraint will be a linear equation and the *join* operator may be the convex hull.

The purpose of restricting the representation of states to an abstract domain is to reduce the computational overhead. Although various abstract domains may be plugged in, the underlying fixed-point computation remains the same. The fixed-point of states are computed on the program's control flow graph (CFG). Without loss of generality, we assume

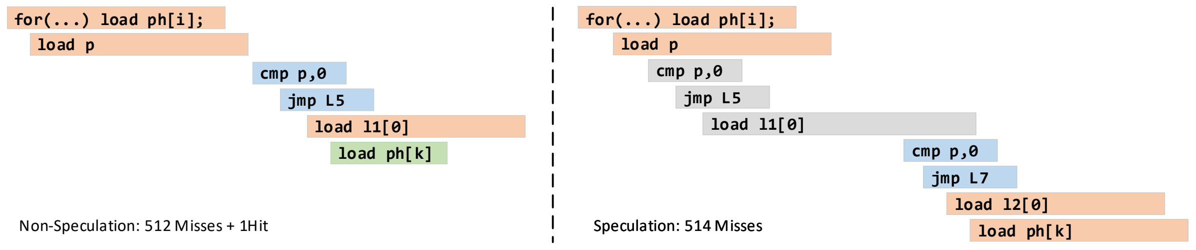

Figure 3. Pipelined execution trace for program in Figure 2

Algorithm 1 Abstract interpretation based static analysis.

```
1: Initialize $S[n]$ to $\top$ if $n = \text{Entry}(CFG)$, and to $\bot$ otherwise
2: $WL \leftarrow \text{Entry}(CFG)$
3: while $\exists\, n \in WL$ do
4:     $WL \leftarrow WL \setminus \{n\}$
5:     $s' \leftarrow \text{Transfer}(S[n], inst_n)$
6:     for all $n' \in \text{Successors}(CFG, n)$ do
7:         if $s' \not\sqsubseteq S[n']$) then
8:             $S[n'] \leftarrow s[n'] \sqcup s'$
9:             $WL \leftarrow WL \cup \{n'\}$
10:        end if
11:    end for
12: end while
```

the CFG has a unique entry node and a unique exit node. Inside the CFG, nodes are associated with instructions or basic blocks of instructions, whereas edges represent the control flows, guarded by conditional expressions.

Let Transfer : $S \times INST \longrightarrow S$ be the transfer function, which takes a state $s \in S$ and an instruction $inst \in INST$ as input, and returns the new state $s' = \text{Transfer}(s, inst)$ as output. s' is the result of executing *inst* in state s.

Algorithm 1 shows a generic procedure that returns, for each CFG node n, an abstract state $S[n]$ as output. $S[n]$ is supposed to be a sound over-approximation of all the possible states at n, regardless of the input values or paths taken to reach n. Initially, $S[n]$ is $\top$ (tautology) for the entry node but $\bot$ (empty) for all other CFG nodes. The remaining part of the procedure is a standard worklist-based algorithm for computing the fixed point [42]: starting from the entry node, it computes the states of the successor nodes (n') based on the transfer function. To ensure convergence, e.g., when the program has loops or is otherwise non-terminating, a *widening* operator (∇) is needed in addition to *join* ($\sqcup$). However, for brevity, we omit the details; for a complete introduction, refer to [14, 40].

The actual definitions of abstract state S and transfer function Transfer depend on the application. In this work, we are concerned with the cache state corresponding to a program. We will present our definitions in Section 4.

3.2 Cache and Speculative Execution

Cache is a type of small but fast storage to hold frequently used data so that they do not need to be fetched from or stored to the large but slow memory every time. Although this work focuses on the data cache, which is more relevant to our applications, the underlying technique can be extended to the instruction cache as well.

In a typical CPU, e.g., an Intel processor [1], instructions are fetched from memory and decoded continuously before they are sent to the scheduler for execution. Executing an instruction involves multiple units; speculative execution [53] is an optimization that efficiently utilizes these execution units. During speculative execution, instructions are scheduled in a pipeline as soon as the required execution units are available; for example, while an instruction is waiting for data to be fetched from memory, subsequent instructions may be executed, as long as the program semantics remains the same to observers from outside of the CPU.

Things become complicated when there are branches, however, since the branch prediction unit must make a guess on which branch target to execute. Instructions in the predicted branch will be executed while the branch condition is being evaluated, and will be committed only after the prediction is confirmed to be correct. Upon misprediction, however, the result of speculative execution will be discarded and the execution will be redirected to the correct branch.

The reorder buffer inside the execution unit, among others, is responsible for this *rollback*: upon a branch mis-prediction, it will not perform register retiring as in a normal execution; instead, it will flush out the affected registers, before restoring the CPU to a previously saved state.

The branch predictor also plays an important role in speculative execution since its accuracy is directly related to the performance of the CPU. However, regardless of the underlying strategies [27, 56, 59], when a branch prediction turns out to be incorrect, the speculatively executed instructions may leave side-effects on the states of other system components, including the cache. In this work, we are concerned with modeling of such side-effects in abstract interpretation.

4 Static Cache Analysis

In this section, we present our instantiation of the baseline abstract interpretation algorithm. The goal is to decide, at each program location, whether a memory access always results in a cache hit. Previously, such *must-hit* analyses were used in execution time estimation [19, 20] and side

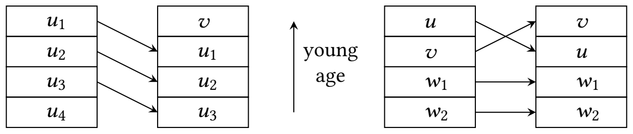

Figure 4. Transfer of the cache state under the LRU policy.

channel mitigation [7, 16, 58]; however, they did not model speculative execution.

4.1 The Abstract State

Let $V = \{v_1, ..., v_n\}$ be the set of program variables stored in memory. Each variable $v \in V$ may be mapped to a cache line. Let the cache be fully associative with the LRU replacement policy, which means a variable $v \in V$ may be mapped to any cache line and, if there is not enough space, the least recently used (LRU) variable will be evicted from the cache. Assume that N is the total number of cache lines, we can define the age of each variable $v \in V$, denoted $Age(v)$, which is an integer ranging from 1 to $N + 1$. Here, $Age(v) = 1$ means v resides in the most recently used line, $Age(v) = N$ means v resides in the least recently used cache line, and $Age(v) = N + 1$ means v is outside of the cache.

The cache state S associated with the entire program is defined as $S = \langle Age(v_1), \ldots, Age(v_n)\rangle$. In this context, a *Must-Hit* analysis needs to compute, at each program location, an upper bound of $Age(v)$. If the upper bound is less than or equal to N, then v must be in the cache. Otherwise, it is possible that v may be outside of the cache.

4.2 The Transfer Function

Let $\textsc{Transfer}(S, inst)$ be the transfer function that models the impact of executing *inst* in the cache state S: given the current state $S = \langle Age(v_1), \ldots, Age(v_n)\rangle$, it returns a new state $S' = \langle Age'(v_1), \ldots, Age'(v_n)\rangle$. If *inst* does not access memory at all, then $S' = S$. Otherwise, assume that $v \in V$ is the variable being accessed in *inst*, and we compute the new state S' as follows:

- For the accessed variable v, set $Age'(v) = 1$ in S'.
- For variable $u \in V$ whose age may be younger than v in S, increment the age of u; that is, $Age(u) < Age(v) \rightarrow Age'(u) = Age(u) + 1$.
- For any other variable $w \in V$, set $Age'(w) = Age(w)$.

Given the definition of $\textsc{Transfer}$ for an instruction, we define it for a sequence of instructions $Insts = \{inst_0, inst_1, ...inst_n\}$ as follows: $\textsc{Transfer}(S, Insts) =$

$$\textsc{Transfer}(\ldots(\textsc{Transfer}(S, inst_0), inst_1), \ldots, inst_n).$$

Figure 4 show two examples. The left-hand-side example illustrates the access of v, which is not yet loaded into the cache. After the access, $Age(v) = 1$, meaning v is loaded to the youngest cache line. Furthermore, the ages of all other lines increase by 1. Since $Age(u_4) > 4$, the variable u_4 is evicted from the cache.

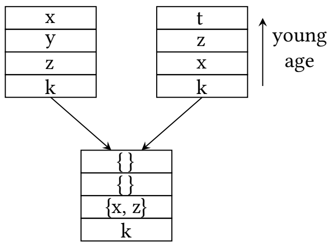

Figure 5. Join of two states at a control-flow merge point.

In the right-hand-side example, however, v is in the cache prior to the execution of the instruction. Thus, existing cache lines fall into two categories. For the variable (u) whose age used to be younger than that of v, the age increases by 1. For the variables (w_1 and w_2) whose ages used to be older than that of v, the ages remain the same.

4.3 The Join Operator

For efficiency reasons, states computed along two program paths are joined together at the control-flow merge point, to avoid creating an exponential number of states. In the baseline abstract interpretation algorithm, the join operator ($\sqcup$) always maintains a single cache state in the result, regardless of how many states are joined.

Figure 5 illustrates a join operator that computes, for each variable $v \in V$, the maximum possible age. For example, the ages of variable x were 1 in the state on the left and 3 in the state on the right; thus, the age of x after join is 3. Similarly, for variable z, the ages were 3 and 1; thus, the age after join is 3. However, for k, since its ages were 4 and 4, after join, the age remains 4. We define the join operator in this way because our goal is to conduct a *Must-Hit* analysis: we know that a variable $v \in V$ is definitely in the cache *only if* v is in the cache according to both states before the join, i.e., $Age(v) \le N$ and $Age'(v) \le N$.

Formally, given two states $S = \langle Age(v_1), \ldots, Age(v_n)\rangle$ and $S' = \langle Age(v'_1), \ldots, Age(v'_n)\rangle$, we define $S'' = S \sqcup S'$ as follows: $S'' = \langle max(Age(v_1), Age(v'_1)), \ldots, max(Age(v_n), Age(v'_n))\rangle$.

5 Modeling the Speculative Execution

In this section, we lift the baseline abstract interpretation algorithm so that it can soundly model speculative execution.

5.1 Augmented CFG with Virtual Control Flow

Given the CFG of a program, we first augment it by adding special nodes and edges, to model all possible control flows produced by speculative executions. These implicit control flows, which will be made explicit in our augmented CFG, are called the *virtual control flows.*

A virtual control flow occurs at every *if-else* statement where the branching condition depends on some variables stored in memory. In a normal execution, a branch guarded by a condition (c) is explored only when c is satisfied. However, under speculative execution, the branch will be explored (speculatively) by our algorithm even if c is unsatisfiable. Furthermore, upon mis-prediction, the rollback will re-direct the control to the other branch.

To model all these behaviors, we add the following special nodes and edges to the CFG for every branch that may be explored speculatively:

- vn_{start}, which is a special CFG node that denotes the start of a virtual control flow;
- vn_{stop}, which is a special CFG node that denotes the end of a virtual control flow.

The edges connecting such nodes, which represent the virtual control flows, fall into five categories: (1) n–vn_{start}; (2) vn_{start}–n; (3) n–n; (4) n–vn_{stop}, and (5) vn_{stop}–n, where n is a normal CFG node.

The edge n–vn_{start} represents the start of a speculative execution: it feeds the state $S[n]$ to vn_{start}, which in turn generates a speculative state $SS[vn_{start}] = S[n]$. Then, the newly created speculative state is propagated through the edge vn_{start}–n. Next, it is propagated through the edges n–n and n–vn_{stop} until reaching vn_{stop}–n. The special node vn_{stop} converts the speculative state $SS[vn_{stop}]$ back to the normal state $S[n] = SS[vn_{stop}]$. Afterward, the state is joined with other states from the normal execution.

One way to add the special nodes and edges is illustrated in Figure 6a. Specifically, for each *if-else* statement, we add virtual control flow edges from instructions in one of the branch to the entry node of the other branch under the same branching condition.

Here, the blue solid lines represent normal executions, whereas the red dashed lines represent virtual control flows associated with speculative executions of the *else*-branch. Virtual control flows associated with the *then*-branch are similar, but omitted in the figure for clarity. The reason why there are more than one dashed lines is because the roll-back point (i.e., location where roll-back occurs) is non-deterministic; to be conservative, we assume it may occur at any moment within the maximum speculation depth.

In practice, the *speculation depth* is platform-dependent and bounded by a few factors [21, 45], e.g., the size of the reorder buffer; the maximum number of unresolved branches that the CPU can handle before it stalls; whether there are division-by-zero or floating-point errors in the program; and the number of clock cycles taken to access memory and resolve a branching condition. For simplicity, for example, we assume that the maximum speculative execution depth is provided by the user. In Figure 6a, we assume that $inst_B$ is the boundary within which roll-back occurs.

5.2 Merging the Speculative Flows

Since we use abstract interpretation to over-approximate the cache states, multiple executions must be merged to reduce the computational overhead. In the baseline algorithm, for example, states from two different paths are joined whenever the program paths are merged in the CFG. In the speculative analysis, we also need to decide when to join the normal and the speculative states.

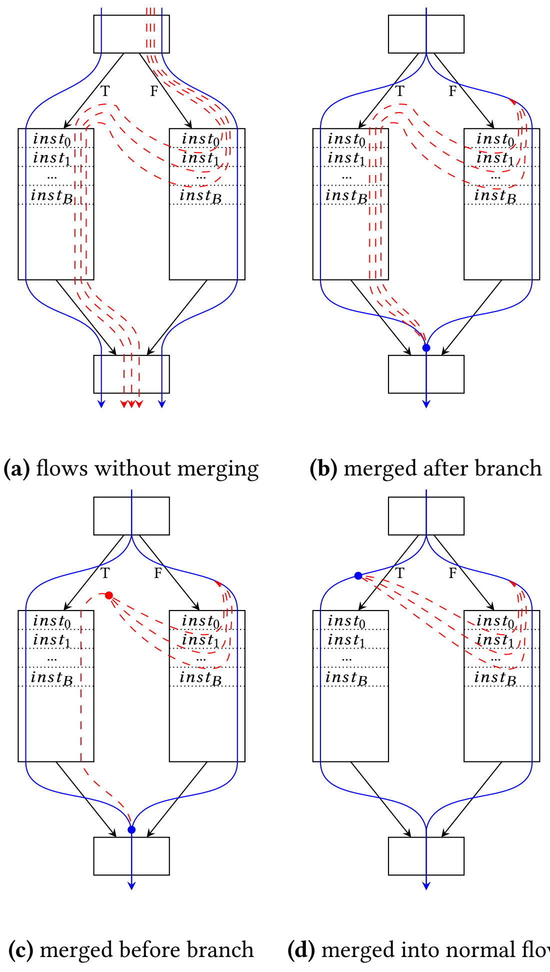

Figure 6. Strategies for merging speculative control flows.

Figure 6 shows three merging strategies in addition to the original *no-merging* strategy in Figure 6a. Consider Figure 6b, for example, since the executions before the branch entry node are identical, they are merged without losing accuracy; in addition, the speculative executions are merged right before the exit point of the other branch. Recall that the join operator ($\sqcup$) used to handle merging is over-approximated, we know that the strategy outlined in Figure 6b is a sound over-approximation of Figure 6a.

To over-approximate even more, consider Figure 6c, which merges all speculative states of the *else*-branch before reaching the *then*-branch. However, the merged speculative state is propagated through the *then*-branch before it is merged with the normal state. In contrast, Figure 6d is a more aggressive over-approximation, which merges the speculative states with the non-speculative state at the entry node of the *then*-branch.

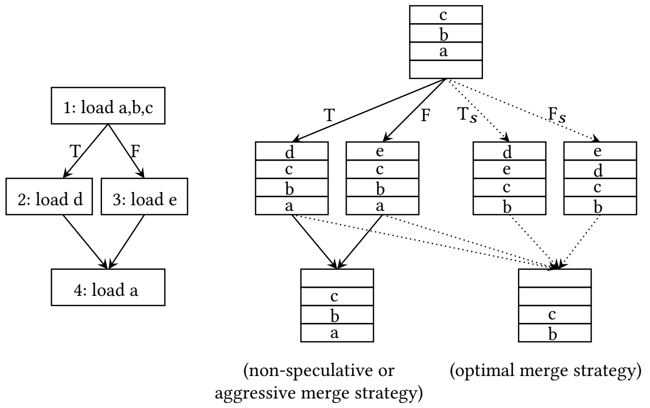

Figure 7. Cache state with different merge points.

Regardless of the merging strategy, however, our method ensures that the result is a sound over-approximation. Since every time state merging occurs, it may lose information, in general, the later that merging occurs, the more accurate the result is, but there is no guarantee. Furthermore, late merging may lead to a more expensive analysis. Our experimental comparisons of these four strategies show that the one outlined in Figure 6c is the best: it not only obtains significantly more accurate results than the one in Figure 6d, but also runs almost equally fast. Therefore, we have settled down on this strategy: we call it *Just-in-Time* merging.

5.3 Just-in-Time Merging: An Example

Consider the CFG of a branch shown on the left-hand side of Figure 7, where each basic block refers to a variable (from a to e). The initial cache state, at the top of the figure on the right-hand side, is the state after executing the first basic block, where variables a, b and c are loaded into the cache. Here, the solid arrows represent the normal execution, where either d or e is mapped to the youngest cache line. Since we are concerned with a *Must-Hit* analysis, after merging at basic block 4, only a, b and c are left in the cache.

Under speculative execution, we may execute the *else*-branch before rolling back to execute the *then*-branch. If we choose to merge the speculative state right after the rollback, the merging would be between d, c, b and a on the one hand, and e, d, c and b on the other hand. The merged state will not contain e anymore, thus losing the important information of speculative execution.

However, if we propagate the speculative state computed from the *else*-branch through the *then*-branch and then merge with the non-speculative state, the cache state at basic block 4 will be more accurate. As shown by the dotted arrow T_s, variable e is loaded to the cache before d is loaded to the cache; similarly, for F_s, variable d is loaded before e is loaded. Finally, when the four states are merged, the result is that only c and b are guaranteed to result in cache hits. Thus, the cache state on the bottom-right of Figure 7, which corresponds to *Just-in-Time* merging illustrated in Figure 6c, captures the side effect of speculative execution.

6 Generalization and Optimization

In this section, we present the generalized algorithm before discussing several optimizations, which help increase accuracy as well as decrease runtime overhead.

Algorithm 2 shows the static analysis procedure that is sound under speculative execution. Given the original CFG of a program, it first constructs an augmented CFG by adding the virtual control flows. Then, it initializes the abstract states for each program location n, including both the default state, denoted $S[n]$, and the speculative state, denoted $SS[n]$. Next, it starts the fixed-point computation using a worklist based procedure that is similar to that of Algorithm 1.

Algorithm 2 Abstract interpretation under speculation.

```
1: $VCFG \leftarrow$ AugmentCFGWithVirtualControlFlow($CFG$)
2: Initialize $S[n]$ to $\top$ if $n \in$ Entry($VCFG$), else to $\bot$
3: Initialize $SS[n]$ to $\bot$ for all $n \in VCFG$
4: $WL \leftarrow$ Entry($VCFG$)
5: while $\exists n \in WL$ do
6:     $WL \leftarrow WL \setminus \{n\}$
7:     if $n$ is a normal CFG node then
8:         $s' \leftarrow$ Transfer( $S[n]$, $n$)
9:         $ss' \leftarrow$ Transfer( $SS[n]$, $n$)
10:    else
11:        Set $ss'$ to $S[n]$ if $n$ is a special $n_{start}$ node, else to $\bot$
12:        Set $s'$ to $SS[n]$ if $n$ is a special $n_{stop}$ node, else to $\bot$
13:    end if
14:    for each $n' \in$ Successors($VCFG$, $n$) do
15:        if $s' \not\sqsubseteq S[n']$ or $ss' \not\sqsubseteq SS[n']$ then
16:            $S[n'] \leftarrow S[n'] \sqcup s'$
17:            $SS[n'] \leftarrow SS(n') \sqcup ss'$
18:            $WL \leftarrow WL \cup \{n'\}$
19:        end if
20:    end for
21: end while
```

However, when the special CFG node vn_{start} is encountered (Line 11), the default state $S[n]$, which is from the incoming edge, is used to create a speculative state $ss' \leftarrow S[n]$; this is to model the side effects caused by the failed speculative execution upon rollback. From then on, both the default state $S[n]$ and the speculative state $SS[n]$ will be propagated through subsequent nodes in the VCFG; at each node n, the transfer function has to be applied to both of them (Lines 8-9). This continues until the other special node vn_{stop} is encountered, which transforms the speculative state $SS[n]$ back to s' (Line 12) before s' is merged into the normal flow.

6.1 The Running Example

To illustrate how the algorithm works, consider the example program in Figure 8, which is a real-time DSP program written in C [25]. The corresponding CFG is shown in Figure 9, where the red (solid and dashed) edges represent the two virtual control flows.

```
/* table is 31-byte long to make quantl look-up
easier, last entry is for mil=30 when wd is max */
int quant26bt_pos[31] = { 61,60,59,58,57,56,55,54,
  53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,
  38,37,36,35,34,33,32,32 };
/* table is 31-byte long to make quantl look-up
easier, last entry is for mil=30 when wd is max */
int quant26bt_neg[31] = { 63,62,31,30,29,28,27,26,
  25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,
  9,8,7,6,5,4,4 };
/* decision levels - pre-multiplied by 8 */
int decis_levl[30] = { 280,576,880,1200,1520,1864,
  2208,2584,2960,3376,3784,4240,4696,5200,5712,
  6288,6864,7520,8184,8968,9752,10712,11664,12896,
  14120,15840,17560,20456,23352,32767 };

int quantl(int el,int detl) {
 int ril,mil;
 long int wd,decis;
 /* abs of difference signal */
 wd = my_abs(el);
 /* mil based on decision levels and detl gain */
 for(mil = 0 ; mil < 30 ; mil++) {
   decis = (decis_levl[mil]*(long)detl) >> 15L;
   if(wd <= decis) break;
 }
 /*if mil=30, wd is less than all decision levels*/
 if(el >= 0) ril = quant26bt_pos[mil];
 else ril = quant26bt_neg[mil];
 return(ril);
}
```

Figure 8. Code snippet from a real-time DSP program [25].

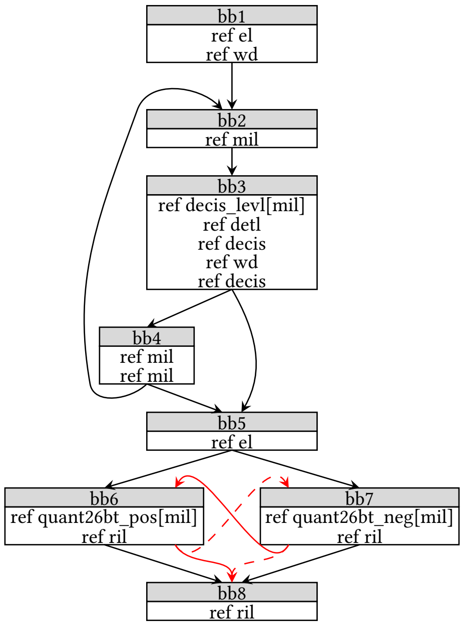

Figure 9. Augmented CFG with virtual control flows.

Table 1. Cache states during the fixed-point computation.

BBlk	Cache State
0	{ }
1	{wd, el}
2	{mil, wd, el}
3	{decis, wd, detl, decis_lev[1*], mil, el}
4	{mil,decis, wd, detl, decis_lev[1*], el}
2	{mil,decis, wd, detl, decis_lev[1*], el}
3	{decis, wd,detl, decis_lev[2*], mil, decis_lev[1*], el}
4	{mil, decis, wd,detl, decis_lev[2*], decis_lev[1*], el}
2	{mil, decis, wd,detl, decis_lev[2*], decis_lev[1*], el}
5	{el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
6	{ril, quant26bt_pos[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
7	{ril, quant26bt_neg[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
8	{ril, ∅ , el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}

Result from Non-speculative Executions Table 1 shows the cache state computed for each location (basic block) based on only normal executions (black edges in Figure 9); this is analogous to running the baseline procedure in Algorithm 1. In Column 2, the variables are arranged according to their ages: the younger variable appears on the left.

Initially, the cache is empty. From basic block 1 to 5, we apply the transfer functions: `decis_lev` takes two cache lines, but since we do not unwind the loop, we do not know its index statically. Thus, we nondeterministically pick one for the first time, `decis_lev[1*]`. Following the back edge from basic block 4, when `decis_lev` is accessed again, we conservatively choose the second cache line for `decis_lev[2*]` to ensure that the cache state remains an over-approximation. Our analysis iterates through the loop three times before it reaches a fixed-point (light gray row) and terminates.

Result from Speculative Executions Table 2 shows the cache state computed under speculative execution. For clarity, we only focus on the cache states relevant to the speculative executions starting from basic block 5. We use two different colors, blue and red, to differentiate the cache states computed from non-speculative (blue) and speculative (red) executions. By considering speculative executions, it is possible for us to access both `quant26bt_pos` and `quant26bt_neg` in a single execution.

Execution Time Estimation The last row of Table 2, which differs from the last row of Table 1, shows that most of the program variables have older ages than before. This is dangerous because, if the cache is only large enough to hold the first eight variables, there will be an additional cache miss, which may force the program to miss its deadline.

Side Channel Detection The additional cache miss may also lead to side-channel leaks. Figure 10 shows a client program that uses the program in Figure 8. The application first accepts some input from the user, then processes it using *quantl* as a subroutine, and finally encrypts the result using a cipher such as AES. Before calling *quantl*, a look-up table named *sbox* is loaded; the lookup table will be used by the cipher while it encrypts the data, during which time a secret *key* is used as the index to access *sbox*.

Table 2. Cache states during speculative execution.

BBlk	Cache State
...	...
5	{el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
6	{ril, quant26bt_pos[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
7	{ril, quant26bt_neg[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
6	{ril, quant26bt_pos[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
7	{ril, quant26bt_neg[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
7	{ril, quant26bt_neg[1*], quant26bt_pos[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
6	{ril, quant26bt_pos[1*], quant26bt_neg[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
8	{ril, ∅ , el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
8	{ril, ∅ , ∅, el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}

```c
#define BUF_SIZE 1024*16
const uint8_t sbox[256] = { 0x63, 0x7c, 0x77,
0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
0x2b, 0xfe, 0xd7, 0xab, 0x76, ... };
int main()
{
 uint32_t inBuf[BUF_SIZE];
 int el, delt, tmp;
 for(int i=0; i< 256; i++) // preload sbox
  tmp = sbox[i];
 for(int i=0; i< BUF_SIZE; i++) // read inBuf
  tmp = inBuf[i];
 tmp = quantl(el, delt);
 AES_encode(inBuf);
}
```

Figure 10. The client code that leads to side-channel leaks.

By controlling the input size, a malicious user can force part of the *sbox* to be evicted from the cache. As a result, for some *key* values, accessing *sbox* results in a cache hit, but for other *key* values, it results in a cache miss. Although timing side channels have been investigated before [7, 16, 24, 58], these prior works never considered speculative execution. Our contribution, in this context, is to show that even if a program is *leak-free* under normal execution, it may still be leaky under speculative execution.

6.2 Dynamically Bounding Speculation Depth

Although the maximum number of speculatively executed instructions is used to construct the augmented CFG, in practice, the number of speculatively executed instructions can be smaller. For example, when all variables needed to resolve a branching condition are in the cache, speculative execution may be shortened. Since our cache analysis aims to decide whether a variable access is a *must-hit*, as the analysis continues it may report more *must-hit* variables, which can be used to bound the speculation depths of other branches.

Thus, we propose an optimization that leverages the *must-hit* variables to dynamically remove virtual control flows that are deemed redundant. Toward this end, we maintain two predefined bounds for each speculative execution, b_h and b_m, which correspond to the branching condition being a cache hit or miss. (Since b_h and b_m are platform-dependent, they are set based on input from the user.) By default, we use b_m as the bound; but as soon as the branching condition is proved to be a must-hit, we switch the bound to b_h.

This optimization not only decreases the computational overhead, i.e., by reducing the number of edges in the VCFG, but also increases the accuracy since it results in a potentially tighter over-approximation. In the extreme case where $b_h = 0$, for example, switching to b_h means avoiding speculative execution all together, which can avoid many bogus behaviors.

While our focus here is on exploiting changes to the speculation depth due to cache misses, the proposed technique may be extended to exploit other sources of changes, e.g., execution units being busy, or division taking a longer time based on the operands.

6.3 Handling the Merges and Loops

The algorithm presented so far uses the join operator ($\sqcup$) to over-approximate the union of two abstract states. However, in the presence of loops, it may have limitations: (1) the resulting state may not be accurate enough, and (2) it may take a long time (or forever) to reach a fixed point.

Thus, we add a widening operator [15] to the standard join operation $s[n'] \sqcup s'$; that is, we use $(s[n'] \sqcup s')\nabla s'$ instead of $s[n'] \sqcup s'$. The idea behind widening (∇) is simple: first, we identify the *direction of growth* from the state s' to the state $(s[n'] \sqcup s')$; then, we over-approximate $(s[n'] \sqcup s')$ in such a way that it maximizes the progress along the *direction of growth*. In the interval domain, for example, if the previous state is $s' = 0 \leq x \leq 3$ and the current state is $s = 0 \leq x \leq 5$, the result of widening would be $s\nabla s' = 0 \leq x \leq +\infty$. To achieve better accuracy, loops with fixed iteration number will be fully unrolled; only unresolved loops will be widened.

Figure 11 shows another loop-related problem. First, variable a is loaded into the cache. Then, inside the loop, every time the branch is executed, $Age(a)$ increases by 1. After the join, however, neither b nor c will be in the cache. Thus, eventually, a is evicted from the cache as well. This is not accurate because, during the actual execution, a will never be evicted. With a refined join operator, we will be able to avoid this problem.

We refine the join operator ($\sqcup$) by adding extra information into the cache state. Similar to Touzeau et al. [54], we introduce a shadow variable $\exists v$ for each $v \in V$. Whenever two states are merged and v appears in only one of the two states, the shadow variable $\exists v$ will remain in the merged cache (while the normal variable v will not). Figure 12 shows an example, where both b and e will be replaced with $\exists b$ and $\exists e$ in the final cache state. That is, there exists a path in which variable b or c is cached.

We also revise the transfer function: the shadow variable $\exists v$ will be removed if a concrete reference to v is applied. For example, in Figure 12, if the variable b is accessed on the merged state, $\exists b$ will be removed from final state.

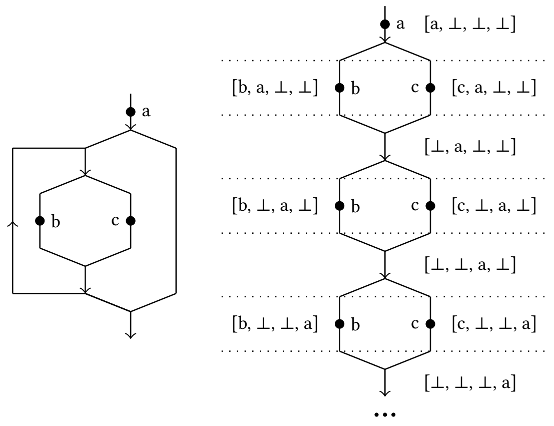

Figure 11. Example program for the widening operator.

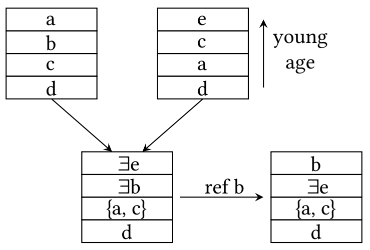

Figure 12. Transfer function with shadow variables.

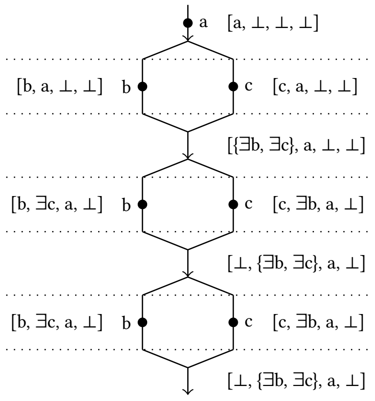

Figure 13. The refined join using shadow variables.

For simplicity, we unroll the loop for three times and illustrate the sequence of memory accesses in Figure 13. The abstract cache states are listed on both sides at each memory access and merge point. With the shadow variables, our cache states are able to reach the fixed-points after only three iterations and avoid evicting a.

6.4 Handling Multiple Speculative Executions

Finally, we extend our algorithm so it can independently propagate the speculative states through the virtual control flows, without interfering each other, even if one branching statement is embedded inside another branching statement.

Algorithm 3 Analysis under a set of speculative executions.

```
1: $(VCFG, C) \leftarrow$ AugmentCFGwithVirtualControlFlow($CFG$)
2: Initialize $S[n]$ to $\top$ if $n \in$ Entry($VCFG$), else to $\bot$
3: Initialize $SS[n][c]$ to $\bot$ for all $n \in VCFG$ and for all color $c \in C$
4: $WL \leftarrow$ Entry($VCFG$)
5: while $\exists n \in WL$ do
6:     $WL \leftarrow WL \setminus \{n\}$
7:     if $n$ is a normal CFG node then
8:         $s' \leftarrow$ Transfer( $S[n]$, $n$)
9:         $ss'[c] \leftarrow$ Transfer( $SS[n][c]$, $n$) for all color $c \in C$
10:    else
11:        Set $s'$ to $SS[n][c]$ if $n$ is node $n_{start}$ of color $c$, else to $\bot$
12:        Set $ss'[c]$ to $S[n]$ if $n$ is node $n_{stop}$ of color $c$, else to $\bot$
13:    end if
14:    for each $n' \in$ Successors($VCFG$, $n$) do
15:        if $s' \not\sqsubseteq S[n']$ or $\exists c : ss'[c] \not\sqsubseteq SS[n'][c]$ then
16:            $S[n'] \leftarrow S[n'] \sqcup s'$
17:            $SS[n'][c] \leftarrow SS(n') \sqcup ss'[c]$ for all color $c \in C$
18:            $WL \leftarrow WL \cup \{n'\}$
19:        end if
20:    end for
21: end while
```

Algorithm 3 shows the procedure, which computes, for each node n in the augmented CFG, a set of states of the form $SS[n][c]$, one for each speculative execution. Let $C = \{1, \ldots, k\}$ be the set of all branches in the program that can be speculatively executed; each $1 \leq i \leq k$ is the index of a branch in this set. We call $c = i$ the color of the i-th speculative execution. While constructing the VCFG, for each $c \in C$, we add a separate set of virtual control-flow edges and nodes, with the color c.

During the fixed-point computation, instead of applying the transfer function once to generate a speculative state ss', the procedure applies the transfer function $|C|$ times, to generate a vector of speculative states $ss'[c]$, one for each speculative execution with color c. As such, every speculative execution (of color $c \in C$) is handled separately until the corresponding node n_{stop} (of the same color c) is encountered, in which case the speculative state $SS[n][c]$ is transformed back to a non-speculative state s'.

There are alternative ways of presenting the analysis procedure in Algorithm 3, for example, by using the trace partitioning framework developed by Mauborgne and Rival [38]. Also note that, for ease of comprehension, we choose to split the speculative states from the normal states. However, the two types of states may be treated uniformly and processed using a generalized worklist-based algorithm. Assume that the worklist-based algorithm is smart enough, the special merge nodes created for virtual control flows can be viewed as merely optimization hints.

7 Experiments

We have implemented our method in LLVM [32] and experimentally compared it with a state-of-the-art, non-speculative static cache analysis technique [58]. In our experiments, we used a set-associative cache with the LRU replacement policy,

Table 3. Execution time estimation: benchmark statistics.

Name	Source	Description	Loc
adpcm	WCET@mdh	motor control	910
susan	MiBench	image process algorithm	2,140
layer3	MiBench	mp3 audio lib	2,233
jcmarker	MiBench	jpeg compose algorithm	1,444
jdmarker	MiBench	jpeg decompose algorithm	2,068
jcphuff	MiBench	jpeg Huffman entropy encoding routines	694
gtk	MiBench	GTK plotting routines	949
g72	mediaBench	routines for G.721 and G.723 conversions	608
vga	mediaBench	Driver for Borland Graphics Interface	386
stc	mediaBench	pson Stylus-Color Printer-Driver	492

Table 4. Side channel detection: benchmark statistics.

Name	Source	Description	Loc
hash	hpn-ssh	hash function	320
encoder	LibTomCrypt	hex encode a string	134
chacha20	LibTomCrypt	chacha20poly1305 cipher	776
ocb	LibTomCrypt	OCB implementation	377
aes	LibTomCrypt	AES implementation	1,838
str2key	openssl	key prepare for des	385
des	openssl	des cipher	1,051
seed	linux-tegra	seed cipher	487
camellia	linux-tegra	camellia cipher	1,324
salsa	linux-tegra	Salsa20 stream cipher	279

512 cache lines, and 64 bytes per line. The speculative execution depths, following a cache hit and a cache miss, are set to 20 and 200 instructions, respectively. These bounds were derived from our analysis of the pipelined execution traces produced by GEM5 [8], a state-of-the-art micro-architecture simulator, with *O3CPU*, which is a detailed out-of-order CPU model based on the Alpha 21264 processor.

Our experiments were designed to answer three questions: (1) Is our method more accurate in detecting cache misses than the existing method, which does not consider speculative execution? (2) Is our method fast enough for practical use? (3) Are the optimizations proposed in Section 6 effective in reducing overhead and increasing accuracy?

7.1 Benchmarks

Tables 3 and 4 show the statistics of our benchmarks, collected from various sources including the *Malardalen* real-time software benchmark [25], a commercially representative embedded software suite named *MiBench* [26], a high performance patch for SSH (*hpn-ssh*) [12], a cryptographic toolkit named *LibTomCrypt* [2], the *openssh* source code [3], and a Linux kernel for *tegra* [4] used on Tesla automobiles. These benchmarks are divided into two sets: execution time estimation and side channel detection. The benchmarks for execution time estimation (Table 3) are used as is, whereas the benchmarks for side channel detection (Table 4) are used together with a client program that we wrote, to invoke the benchmark program in a way similar to Figure 10.

7.2 Effectiveness: Execution Time Estimation

We first compare our method with the state-of-the-art, non-speculative method [58]. The results are shown in Table 5. For our method, we also report the number of speculative cache

Table 5. Execution time estimation: comparisons in terms of the analysis time and the number of cache misses.

Name	Non-speculative		Speculative				
	Time (s)	#Miss	Time (s)	#Miss	#SpMiss	#Branch	#Iteration
adpcm	0.98	24	12.70	32	17	75	173
susan	19.40	17	248.40	26	17	113	464
layer3	7.24	78	65.54	88	35	241	374
jcmarker	0.20	22	3.40	26	11	37	72
jdmarker	2.89	21	15.18	78	55	193	726
jcphuff	0.03	12	0.44	12	13	25	32
gtk	19.90	16	274.76	19	13	77	190
g72	0.16	6	0.94	9	4	41	79
vga	0.05	4	0.06	4	3	3	3
stc	0.13	10	0.96	23	14	39	105

Table 6. Execution time estimation: comparisons of two strategies for merging speculative executions.

Name	Merging at rollback point				Just-in-time merging			
	Time(s)	#Miss	#SpMiss	#Ite	Time(s)	#Miss	#SpMiss	#Ite
adpcm	14.40	31	25	261	12.70	32	17	173
susan	405.70	30	29	620	248.40	26	17	464
layer3	84.64	94	53	471	65.54	88	35	374
jcmarker	4.80	27	19	99	3.40	26	11	72
jdmarker	16.11	35	59	777	15.18	78	55	726
jcphuff	0.48	12	10	36	0.44	12	13	32
gtk	358.56	24	26	236	274.76	19	13	190
g72	1.28	7	1	122	0.94	9	4	79
vga	0.07	4	3	5	0.06	4	3	3
stc	1.86	31	35	222	0.96	23	14	105

misses (#SpMiss), which are not observable from outside of the CPU, the number of conditional branches that can be speculatively executed, and the total number of iterations of our method on loops.

The results show that our method detected more cache misses, thus highlighting the unsoundness of the existing method and the importance of modeling speculative execution during execution time estimation.

As for the analysis time, our method completed all the benchmarks, although it took a longer time than the non-speculative analysis due to its focus on being always sound. The reason why it took significantly longer for the *gtk* benchmark, in particular, is because the program has a large data size (of nearly 3 MB), which led to a large number of variables to be tracked in the abstract cache state.

Table 6 compares two merging strategies in terms of the analysis time, the number of cache misses, the number of speculative cache misses, and the number of iterations. The result is somewhat surprising in that although *merging at rollback point* is more aggressive than *just-in-time merging*, the later is actually faster. The reason is because merging the speculative state with the normal state right after the rollback point may force the normal state to become a coarser-grained over-approximation. This can lead to a slower convergence to a coarser fixed point, as shown by the data in Columns 5 and 9. However, there are exceptions, indicating that *optimal merging* in general is *problem-specific*, and the accuracy depends on the combined effects of branches and loops in a program.

Table 7. Side channel detection: comparisons in terms of the analysis time and whether leaks are detected.

Name	Buffer (byte)	Non-speculative		Speculative	
		Time (s)	Leak Detected	Time (s)	Leak Detected
hash	31,808	0.67	No	1.15	Yes
encoder	32,512	0.03	No	0.10	Yes
chacha20	26,304	1.18	No	9.24	Yes
ocb	31,616	0.10	No	0.68	Yes
aes	32,768	0.08	No	2.13	No
str2key	32,768	0.01	No	0.01	No
des	0	0.60	No	14.20	Yes
seed	32,768	0.01	No	0.07	No
camellia	32,768	0.35	No	6.35	No
salsa	32,768	0.02	No	0.06	No

7.3 Effectiveness: Side Channel Detection

Table 7 shows the results for side channel detection, including comparisons of the two methods in terms of the analysis time and whether leaks are detected. In this context, a leak refers to the dependency between the cache behavioral difference and sensitive data; furthermore, whether there is a leak or not often depends on the input buffer size controlled by the (potentially malicious) user. Thus, during experiments, we set the buffer size to various values from 32K bytes (the size of cache we use) down to 0 byte.

Generally speaking, the larger the buffer size, the easier that the client program triggers the behavioral difference. Thus we first set the buffer size to 32KB, and starting from there we gradually reduce the buffer size and keep track of the impact of speculative execution on cache state, until the two methods return different results.

Since the benchmarks are mostly cryptographic algorithms, which are relatively small in terms of the number of lines of code, the analysis time is short. Furthermore, our method successfully detected leaks in half of the benchmarks, whereas the existing (unsound) method did not detect leaks in any of them. This highlights the importance of having a sound static cache analysis for speculative execution, e.g., to detect more leaks and avoid bogus proofs (that there is no leak). On one of the benchmarks, *des*, leaks are detected even if the buffer size is set to 0 because, even without the client program, the benchmark program itself has a user controlled buffer, which can be set to sizes that induce timing side-channel leaks under speculative execution.

As a static analysis procedure, our method may generate false positives. In addition to abstraction, the other source of false positives is modeling of the speculative execution. Therefore, for each of the new leaks detected by our method in Table 7, we manually inspected the software code and the execution trace. Our inspection confirmed that all of them are indeed real leaks; that is, there exist specific memory/cache layouts and execution traces that induce the leaks.

8 Related Work

Abstract interpretation [14] is a framework for conducting static analysis and proving properties. Ferdinand and Wilhelm [18, 20] pioneered the use of abstract interpretation in may- and must-hit cache analyses [57]. Others also used similar techniques to detect timing side channels [7, 16, 58]. However, prior works focused primarily on improving abstract interpretation without considering speculative execution.

There are some techniques that consider the impact of speculative execution [34], but only for the instruction pipeline. In a commercial tool named *AIT*, speculations are also considered during execution time estimation by leveraging a standalone pipeline analysis as a driver [57]. Since the tool is propriety, details of this analysis have not been made public; therefore, it is not clear how speculative execution is modeled during abstract interpretation.

Our method differs from the large body of work on statistically estimating the worst-case execution time of real-time software [33, 35, 41] using either CPU simulators or characteristics of prior simulation results [52]. These techniques, while useful, are not designed to be sound, and hence may not be suitable for the applications that we have in mind, such as detecting side-channel leaks or proving that leaks do not exist. The reason is because, if the analysis is not sound, the proof may not be valid and as a consequence, leaks may be left undetected.

For timing side channels, many analysis and verification techniques [6, 9, 10, 24, 43, 44, 51, 58] have been developed, including the one proposed by Chen et al. [11], which uses Cartesian Hoare Logic [49] to prove that timing leaks of a program are bounded. Antonopoulos et al. [5] also developed a method for proving the absence of timing channels. However, these methods only consider *instruction-induced* timing variance while ignoring the cache.

There are also techniques for improving the accuracy of cache analysis, e.g., by using symbolic execution or model checking to refine the cache analysis results [13, 39, 54] and by extending the analysis from single-core to multi-core CPUs [37]. However, none of these techniques considered speculative execution, which is the focus of our work.

9 Conclusions

We have presented a new abstract interpretation technique that can soundly analyze a program under speculative execution. The goal is to lift existing static analyzers, which were geared toward analyzing only non-speculative executions, so that they become sound also for speculative executions. We have implemented the technique in a static cache analysis tool and evaluated it on two sets of benchmarks, for execution time estimation and side channel detection. Our experimental results show that the method can detect many cache misses and side-channel leaks overlooked by a state-of-the-art non-speculative analysis technique.

Acknowledgments

This work was partially funded by the U.S. National Science Foundation (NSF) under the grants CNS-1617203 and CNS-1702824 and the Office of Naval Research (ONR) under the grant N00014-17-1-2896.

References

[1] 2014. *Intel ®64 and IA-32 Architectures Optimization Reference Manual.* https://botan.randombit.net/.

[2] 2018. *LibTomCrypt: A Modular and Portable Cryptographic Toolkit.* https://www.libtom.net/LibTomCrypt.

[3] 2018. OpenSSH. http://www.openssh.com/

[4] 2018. *Tesla Motors: Linux.* https://github.com/teslamotors/linux.

[5] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* 362–375.

[6] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. 2016. String analysis for side channels with segmented oracles. In *ACM SIGSOFT Symposium on Foundations of Software Engineering.* 193–204.

[7] Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2014. Leakage resilience against concurrent cache attacks. In *International Conference on Principles of Security and Trust.* 140–158.

[8] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The GEM5 simulator. *SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[9] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2018. Symbolic path cost analysis for side-channel detection. In *International Conference on Software Engineering.* 424–425.

[10] Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. 2017. *String Analysis for Software Verification and Security.*

[11] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *ACM SIGSAC Conference on Computer and Communications Security.* 875–890.

[12] Rapier Chris, Steven Michael, Bennett Benjamin, and Tasota Mike. 2018 (accessed March 1, 2019). High Performance SSH/SCP - HPN-SSH. https://www.psc.edu/hpn-ssh

[13] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. 2016. Precise cache timing analysis via symbolic execution. In *IEEE Real-Time and Embedded Technology and Applications Symposium.* 1–12.

[14] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* 238–252.

[15] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* 84–96.

[16] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Security Symposium.* 431–446.

[17] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. Formal Verification of Software Countermeasures against Side-Channel Attacks. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 11:1–11:24.

[18] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. 1999. Cache behavior prediction by abstract interpretation. *Science of Computer Programming* 35, 2-3 (1999), 163–189.

[19] Christian Ferdinand and Reinhard Wilhelm. 1998. On predicting data cache behavior for real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems.* 16–30.

[20] Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems* 17, 2-3 (1999), 131–181.

[21] Eliseu M. Chaves Filho and Edil S. Tavares Fernandes. 1997. The Effect of the Speculation Depth on the Performance of Superscalar Architectures. In *International Euro-Par Conference on Parallel Processing.* 1061–1065.

[22] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference.* 1:1–1:17.

[23] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering* 8, 1 (2018), 1–27.

[24] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 377–388.

[25] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks – Past, Present and Future. In *International Workshop on Worst-Case Execution Time Analysis.* 137–147.

[26] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization.* 3–14.

[27] Daniel A Jiménez and Calvin Lin. 2001. Dynamic branch prediction with perceptrons. In *IEEE International Symposium On High Performance Computer Architecture.* 197–206.

[28] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[29] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic quantification of cache side-channels. In *International Conference on Computer Aided Verification.* 564–580.

[30] Markus Kusano and Chao Wang. 2016. Flow-Sensitive Composition of Thread-Modular Abstract Interpretation. In *ACM SIGSOFT Symposium on Foundations of Software Engineering.*

[31] Markus Kusano and Chao Wang. 2017. Thread-modular static analysis for relaxed memory models. In *ACM SIGSOFT Symposium on Foundations of Software Engineering.* 337–348.

[32] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE / ACM International Symposium on Code Generation and Optimization.* 75–88.

[33] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. 2003. Accurate timing analysis by modeling caches, speculation and their interaction. In *ACM/IEEE Design Automation Conference.* 466–471.

[34] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. 2006. Modeling out-of-order processors for WCET analysis. *Real-Time Systems* 34, 3 (2006), 195–227.

[35] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2009. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In *IEEE Real-Time Systems Symposium.* 57–67.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[37] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. 2010. Combining abstract interpretation with model checking for timing analysis of multicore software. In *IEEE Real-Time Systems Symposium.* 339–349.

[38] Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *European Symposium on Programming Languages and Systems.* 5–20.

[39] Ravindra Metta, Martin Becker, Prasad Bokil, Samarjit Chakraborty, and R Venkatesh. 2016. TIC: a scalable model checking based approach to WCET estimation. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 72–81.

[40] Antoine Miné. 2006. The octagon abstract domain. *Higher-order and symbolic computation* 19, 1 (2006), 31–100.

[41] Tulika Mitra, Jürgen Teich, and Lothar Thiele. 2018. Time-Critical Systems Design: A Survey. *IEEE Design & Test* 35, 2 (2018), 8–26.

[42] F Nielson, Riis H Nielson, and CL Hankin. 1999. Principles of Program Analysis. (1999).

[43] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *IEEE Computer Security Foundations Symposium*. 387–400.

[44] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *IEEE Computer Security Foundations Symposium*. 328–342.

[45] Jim Pierce and Trevor N. Mudge. 1994. The Effect of Speculative Execution on Cache Performance. In *International Symposium on Parallel Processing*. 172–179.

[46] Jörn Schneider. 1998. Statische pipeline-analyse für echtzeitsysteme. *Dipl. thesis, Univ. Saarland, Saarbruecken, Germany* (1998).

[47] Jörn Schneider and Christian Ferdinand. 1999. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *ACM SIGPLAN Notices*, Vol. 34. 35–44.

[48] Rathijit Sen and Y. N. Srikant. 2007. WCET estimation for executables in the presence of data caches. In *ACM/IEEE International conference on Embedded Software*. 203–212.

[49] Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 57–69.

[50] Chungha Sung, Markus Kusano, and Chao Wang. 2017. Modular verification of interrupt-driven software. In *IEEE/ACM International Conference On Automated Software Engineering*. 206–216.

[51] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: a cache timing analysis framework via LLVM transformation. In *IEEE/ACM International Conference On Automated Software Engineering*. 904–907.

[52] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. 2000. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems* 18, 2 (2000), 157–179.

[53] R. M. Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (Jan 1967), 25–33.

[54] Valentin Touzeau, Claire Maiza, David Monniaux, and Jan Reineke. 2017. Ascertaining uncertainty for efficient exact cache analysis. In *International Conference on Computer Aided Verification*. 22–40.

[55] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.

[56] Lucian N Vintan and Mihaela Iridon. 1999. Towards a high performance neural branch predictor. In *International Joint Conference on Neural Networks*. 868–873.

[57] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. 2010. Static timing analysis for hard real-time systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. 3–22.

[58] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *International Symposium on Software Testing and Analysis*. 15–26.

[59] Tse-Yu Yeh and Yale N Patt. 1991. Two-level adaptive training branch prediction. In *IEEE/ACM International Symposium on Microarchitecture*. 51–61.

[60] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. 2018. SCInfer: Refinement-Based Verification of Software Countermeasures Against Side-Channel Attacks. In *International Conference on Computer Aided Verification*. 157–177.