CANAL: A Cache Timing Analysis Framework via LLVM Transformation

Chungha Sung  
University of Southern California  
Los Angeles, CA, USA

Brandon Paulsen  
University of Southern California  
Los Angeles, CA, USA

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

ABSTRACT

A unified modeling framework for non-functional properties of a program is essential for research in software analysis and verification, since it reduces burdens on individual researchers to implement new approaches and compare existing approaches. We present CANAL, a framework that models the cache behaviors of a program by transforming its intermediate representation in the LLVM compiler. CANAL inserts auxiliary variables and instructions to allow standard verification tools to handle a new class of cache related properties, e.g., for computing the worst-case execution time and detecting side-channel leaks. We demonstrate the effectiveness of CANAL using three verification tools: KLEE, SMACK and Crab-llvm. We confirm the accuracy of our cache model by comparing with CPU cycle-accurate simulation results of GEMS.

CCS CONCEPTS

• Software and its engineering → Software verification and validation; • Security and privacy → Cryptanalysis and other attacks;

KEYWORDS

cache, execution time, side channel, verification, symbolic execution, abstract interpretation, bounded model checking

ACM Reference Format:

1 INTRODUCTION

Analyzing the cache behaviors of a program is important, e.g., for computing the worst-case execution time of a real-time system [5, 15] and detecting information leaks through side channels [8, 15]. However, existing verification tools are often designed only for checking functional properties, e.g., assertions or pre- and post-conditions. For example, none of the participants of recent software verification competitions [2] can verify non-functional properties such as those related to the execution time. Although specialized tools have been developed to handle such non-functional properties, they are rarely open-source or as well-maintained as mainstream verification tools. As a result, it is difficult for individual researchers to implement new approaches for verifying such properties or evaluate existing approaches.

We fill the gap by developing a lightweight cache modeling framework for standard verification tools, by transforming the LLVM intermediate representation (IR) of a program to add self-modeling capabilities. That is, we insert auxiliary variables and LLVM instructions over these variables to record and update cache statistics related to Load/Store instructions during the program execution. By using the instrumented LLVM bitcode as input, standard functional verification tools will have the capability of verifying a new class of (non-functional) properties.

Our modeling framework, named CANAL, takes C/C++ code as input and emits LLVM bitcode as output. Thus, it can be used by any LLVM-based verification tools. For example, symbolic execution tools such as KLEE [4] may take the program instrumented by CANAL to detect side-channel leaks; bounded model checkers such as SMACK [14] may take the program instrumented by CANAL to conduct Must- and May-hit cache analyses; and static analyzers based on numerical abstract interpretation, such as Crab-llvm [10], may take the program instrumented by CANAL to conduct worst-case execution time (WCET) analysis. CANAL is available on GitHub1.

In the remainder of this paper, we shall explain how to combine CANAL with KLEE, SMACK and Crab-llvm to obtain the desired results. We also compare CANAL with the CPU cycle-accurate simulation results of GEMS [9], a standard micro-architectural simulator, to demonstrate the accuracy of our cache model.

2 TOOL OVERVIEW

Figure 1 shows the overall flow of CANAL, which takes the C/C++ code of a program and the cache configuration file of a target computer as input, and returns the instrumented LLVM bitcode as output. After compiling the C/C++ code into LLVM bitcode, it uses a sequence of optimization (opt) passes to insert, before or after each Load/Store instruction, some new instructions that model the change of cache states due to these memory accesses. The inserted instructions can be understood as invocations of two functions: _CSIM_Load(addrInfo) and _CSIM_Store(addrInfo), which updates our model of the cache state whenever Load or Store is executed; addrInfo denotes information of the memory location.

1Tool and benchmarks: https://github.com/canalcache/canal
In addition to the automatically inserted calls to \texttt{__CSIM\_Load} and \texttt{__CSIM\_Store}, the user of CANAL may specify properties using these auxiliary variables: \texttt{__CSIM\_num\_hit}, \texttt{__CSIM\_num\_miss}, \texttt{__CSIM\_Load\_ret}, and \texttt{__CSIM\_Store\_ret}. They represent the cumulative numbers of hits and misses along a program path, as well as the cache status (hit or miss) associated with each memory access. By feeding the instrumented LLVM bitcode to standard verification tools as input, CANAL allows them to verify a new class of non-functional properties, such as assertions over auxiliary variables that model the cache behaviors of the program.

Accuracy. To demonstrate the accuracy of our cache model, we compare our results with the cache statistics reported by GEM5. Toward this end, note that CANAL may be used as a standalone cache simulator: if we compile the LLVM bitcode instrumented by CANAL to an executable and run it with a concrete input, it will produce the cache statistics associated with that particular execution. Figure 2 shows this usage case, where \texttt{__CSIM\_init\_cache()} and \texttt{__CSIM\_print\_stat()} are inserted to the original C program to initialize the cache states and display the result, respectively. The function body of \texttt{aes\_encrypt()} will be instrumented by CANAL automatically.

Since the cache statistics reported by GEM5 include not only the main(\texttt{main}) function but also operating system code executed before and after, we need to create two program versions and then compute their difference. One of the programs consists of the main(\texttt{main}) function and instructions inserted at the beginning and end of the main(\texttt{main}) function to flush the cache, while the other program consists of only these cache-flushing instructions with an empty main(\texttt{main}) function body. By running these two programs and computing the difference, we have obtained the exact numbers of cache hits and misses reported by GEM5.

Table 1: Accuracy comparison: CANAL versus GEM5

<table>
<thead>
<tr>
<th>Name</th>
<th>LoC</th>
<th>Mem.access</th>
<th>GEM5</th>
<th>CANAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ary Acc</td>
<td>70</td>
<td>522,575</td>
<td>3.88</td>
<td>3.90</td>
</tr>
<tr>
<td>BullSort</td>
<td>49</td>
<td>122,902</td>
<td>10.62</td>
<td>10.90</td>
</tr>
<tr>
<td>Distort</td>
<td>49</td>
<td>2,619,985</td>
<td>15.54</td>
<td>15.68</td>
</tr>
<tr>
<td>AES(1)</td>
<td>788</td>
<td>534</td>
<td>0.31</td>
<td>0.31</td>
</tr>
<tr>
<td>DES(1)</td>
<td>368</td>
<td>580</td>
<td>0.11</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 1 shows the comparison of GEM5 and CANAL for five example programs, including three sorting routines and two cryptographic routines. The results are always identical. The sorting routines exhibit a diverse range of memory-accessing behavior. By feeding the instrumented LLVM bitcode to standard verification tools as input, CANAL allows them to verify a new class of non-functional properties, such as assertions over auxiliary variables that model the cache behaviors of the program.

3 APPLICATION SCENARIOS

We now demonstrate how CANAL may be used by KLEE, SMACK, and Crab-llvm using five example programs taken from the S5-COMP benchmark [2]: \texttt{copiesome}, \texttt{sanfoundry}, and \texttt{standard} from the array-programs section, and \texttt{gcd} and \texttt{sum} from the bit-vector section. We use a 4-way associative cache with LRU and 64-byte cache line while setting the cache size to 1 KB, 16 KB and 32 KB, respectively. We set the timeout to one hour for each program.

3.1 Combined with Symbolic Execution Tools

Symbolic execution is a technique for systematically exploring feasible paths of a program and generating their test inputs. Although it has been used primarily for checking functional properties, with CANAL, it can now be used to detect timing side-channel leaks.

Timing Side-channel Leaks. We say that a program \texttt{P}(\texttt{k}) with sensitive input \texttt{k} has timing side-channel leaks if the execution time of \texttt{P} depends on the value of \texttt{k}. That is, \texttt{3k}, \texttt{k} \in dom(\texttt{k}): \tau(P, \texttt{k}) \neq \tau(P, \texttt{k}2), where \texttt{k}1 and \texttt{k}2 are values in the domain of \texttt{k} and \tau is the execution time.

Figure 3 shows an example, where \texttt{input1} and \texttt{input2} are marked as symbolic values and used to run the program \texttt{prog()} twice. After each execution, the numbers of hits and misses are stored in \texttt{h1}, \texttt{m1}, \texttt{h2}, and \texttt{m2}, respectively. Finally, the assertion checks if \texttt{prog()} is leak-free: that is, \forall \texttt{input1}, \texttt{input2}, the condition \texttt{(h1==h2 \&\& m1==m2)} always holds. KLEE can be used to search for concrete values of \texttt{input1} and \texttt{input2} that violate the assertion.

Table 2 shows the results of running KLEE on these programs. In each case, we manually modified the program to mark one or more parameters as the sensitive input. Columns 3-5 show if a leak is detected, together with the total number of tests generated and, among them, the number tests that manifest the leak. Columns 6-8 show the time taken by KLEE for the different cache sizes.

Table 2: Results of the side-channel leak detection.

<table>
<thead>
<tr>
<th>Name</th>
<th>LoC</th>
<th>Detection</th>
<th>Ex. tests</th>
<th>Leak tests</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>copiesome</td>
<td>79</td>
<td>Yes</td>
<td>194</td>
<td>9</td>
<td>1.82</td>
</tr>
<tr>
<td>standard</td>
<td>61</td>
<td>No leak</td>
<td>91</td>
<td>0</td>
<td>3.52</td>
</tr>
<tr>
<td>gcd</td>
<td>52</td>
<td>Leak</td>
<td>6</td>
<td>1</td>
<td>0.41</td>
</tr>
<tr>
<td>sum</td>
<td>56</td>
<td>Leak</td>
<td>6</td>
<td>1</td>
<td>0.97</td>
</tr>
</tbody>
</table>
3.3 Combined with Static Analysis Tools

Static analyzers based on numerical abstract interpretation [7], such as Crab-llvm, can generate program invariants. These invariants, computed for each program location, are summaries over all paths and input values. Therefore, they can be used to estimate the worst-case execution time of a program. More specifically, by leveraging CANAL, tools such as Crab-llvm can generate invariants in terms of auxiliary variables such as (5 ≤ _CSIM_num_Load_hit ≤ 18).

Figure 5 shows an if-else statement controlled by the value of cond. Assume each cache line contains 64 bytes, the first 16 integers of the array fall into one cache line, whereas the next 16 integers starting with buffer[16] fall into another cache line. However, during static analysis, there is no way of knowing what the value of cond is; therefore, one has to assume both branches may be taken. When the Then-branch is taken, buffer[5] will be loaded to the cache, which means the access to buffer[5] is a cache hit. However, when the Else-branch is taken, buffer[5] will not be loaded to the cache, which means the access to buffer[5] is a cache miss. By using numerical abstract interpretation, Crab-llvm can take both cases into consideration and compute value ranges of n_s, n_s_h and n_s_m. For this example, in particular, the value ranges would be [2,2] for n_s, [0,1] for n_s_h, and [1,2] for n_s_m. Therefore, Crab-llvm can prove the second and the third assertions, while reporting a potential violation of the first assertion.

In addition, an interesting application of the value ranges computed by numerical abstract interpretation is to compute the worst-case execution time (WCET), which depends on the maximum number of cache misses along all program paths.

Table 4 shows the results of applying CANAL+Crab-llvm on the example programs. Columns 3-6 report the value ranges of the total number Store-hits (S.Hit), Store-misses (S.Miss), Load-hits (L.Hit) and Load-misses (L.Miss). Since these programs have loops and Crab-llvm uses aggressive over-approximation to force termination over loops, most of the upper bounds become infinity. Luckily, these are fixed-bound loops, and after we automatically unrolled these loops, Crab-llvm obtained more accurate value ranges.

### 4 CACHE MODELING

We now briefly explain how cache is modeled inside CANAL. It is a lightweight cache model in that the modeling instructions are carefully designed to reduce the overhead of the verification tools. For example, pointers are difficult to handle by verification tools; therefore, we avoid using them in the instrumented code.

#### 4.1 Pre-computing Address-to-Cache Mapping

Inside LLVM, we first obtain the memory address of each program instruction in the instrumented code. We then, for the target computer architecture, we generate a memory layout. We try to pre-compute the possible address value for each load or store instruction in the program.

If the address is a fixed value, we compute its set and tag fields in the cache, and use these concrete values to simplify the instantiation of _CSIM_Load() and _CSIM_Store(). Otherwise, we resort to the use of if-else statements to dynamically compute the set and tag fields (more difficult to handle by verification tools).

Figure 6 shows a simple case where the address of var is statically known, and thus we can pre-compute its set(242) and tag(1). These concrete values are used to instantiate _CSIM_Load() and _CSIM_Store(). Although C code is used in Figure 6, this is actually implemented at the LLVM bitcode level inside CANAL.

Figure 7 shows a more complex case, where the array is accessed using a variable i, and thus if-else statements are used to compute the set and tag of buffer[1]. Although we cannot simplify as much as in Figure 6, we can still pre-compute the value ranges of set and tag based on the address of the array. In particular, we can assume the value range of set is [0,3] and the tag is always 242.
4.2 Simplifying Updates of the Cache Statistics

To simplify the storage and update of cache statistics so verification tools can handle them easily, we use a set of simple variables as opposed to an array indexed by memory addresses. This can drastically reduce the complexity of the cache-modeling instructions inside functions __CSIM_Load() and __CSIM_Store().

Figure 7 shows the internals of __CSIM_Store(), which updates the cache statistics based on the values of set and tag. Instead of using monolithic arrays such as cacheline[set].tag, we use individual variables such as __CSIM_cache00_tag.

The number 00 means the cache line is associated with set 0 and way 0, and the auxiliary variable denotes the tag saved at the line. When a cache miss occurs, for example, we update the value of cacheline00_tag as well as the values of similar auxiliary variables, and evicts a victim. In this implementation, LRU policy is used to compute the victim; but other replacement policies may be incorporated into CANAL easily.

Implementations of __CSIM_Store() and __CSIM_Load() are specific to each individual program under verification, and therefore they are generated by CANAL automatically.

5 RELATED WORK

CANAL is the first LLVM-based lightweight cache modeling framework designed specifically for software verification tools. Although there are other cache simulators [3, 12] and CPU simulators such as GEM5 [9], they are not designed for this purpose. In particular, they cannot be used in the same way as CANAL to afford existing verification tools the capability of verifying a new class of cache related non-functional properties.

There are also tools designed specifically for WCET analysis based on cache analysis [5, 6, 13] and for detecting cache timing side channels [8, 11, 15]. However, the modeling part of these tools are tied up with the subsequent analysis part, and therefore cannot be used by other verification tools. Furthermore, the analysis part of these tools is rarely open-source, and often not as well-maintained as the mainstream software verification tools, which are updated constantly to keep up with the competition [2].

Although our main contribution in this work is the lightweight cache modeling that facilitates the subsequent analysis and verification, there is still room for improvement in the analysis and verification algorithms. Since cache timing behaviors are non-functional properties, they often have significantly different characteristics from functional properties, and thus may benefit from specialized algorithms to make verification more efficient and scalable.

Our implementation of CANAL has been tested on programs from two domains: real-time software and embedded software. In both cases, the program structure and language constructs are relatively simple. To handle C/C++ programs in other application domains, more sophisticated static analyses may be needed, e.g., to deal with pointer aliasing and complex loops during the pre-computation of address-to-cache mapping and updates of the cache statistics, in order to keep the application of our LLVM based transformation efficient. We also plan to further refine our cache model, e.g., to handle multi-threading as well as multi-level cache.

6 CONCLUSIONS

We have presented CANAL, a framework for modeling cache behaviors of a program based on LLVM transformations, which allows standard software verification tools to check cache timing properties. We demonstrated the accuracy of our cache model in CANAL by comparing with the simulation results of GEM5, as well as the effectiveness of combining CANAL with three existing tools (KLEE, SMACK and Crab-llvm) in verifying cache related properties.

ACKNOWLEDGMENTS

This work was supported in part by NSF under grants CNS-1617203 and CNS-1702824 and ONR under grant N00014-17-1-2896.

REFERENCES