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 over these variables, 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 GEM5. CANAL is available on GitHub[1] and YouTube[2].

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

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, 13] 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.

[1]Tool and benchmarks: https://github.com/canalcache/canal
[2]Demo video: https://youtu.be/JDou3F1j2nY

, ,
.

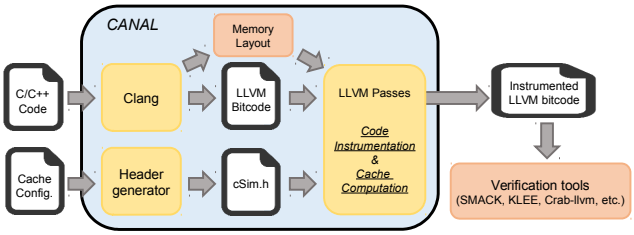

Figure 1: The overall flow of CANAL.

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.

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 GEM5 [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.

In addition to the automatically inserted calls to `__CSIM_Load` and `__CSIM_Store`, the user of CANAL may specify properties using these auxiliary variables: `__CSIM_num_hit`, `__CSIM_num_miss`, `__CSIM_Load_ret`, and `__CSIM_Store_ret`. They represent the accumulative 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 `__CSIM_init_cache()` and `__CSIM_print_stat()` are inserted to the original C program to initialize the cache states and display the result, respectively. The function body of `aes_encrypt()` will be instrumented by CANAL automatically.

```
int main() {
  __CSIM_init_cache();
  char out[16];
  aes_encrypt("hello world!", out);
  __CSIM_print_stat();
}
```

Figure 2: CANAL as standalone cache simulator.

Since the cache statistics reported by GEM5 include not only the `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()` function and instructions inserted at the beginning and end of the `main()` function to flush the cache, while the other program consists of only these cache-flushing instructions with an empty `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 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 based on input data (array of random integers). The cryptographic routines have security-critical computations that are often target of side-channel attacks; their inputs are an encrypted message of "hello world!" using a predefined encryption key. To ensure that we trigger a rich set of cache behaviors during the experiments, we configured the cache to be 4-way associativity with LRU replacement policy, 64-byte line size, and 1K-byte cache size. With a larger cache size, the simulation speed of CANAL will not change much but there will be fewer cache conflicts.

Table 1: Accuracy comparison: CANAL versus GEM5

			GEM5			CANAL		
Name	LoC	Mem.access	R.Miss	W.Miss	Time(s)	R.Miss	W.Miss	Time(s)
Ary Acc	70	322,575	2,881	0	0.62	2,881	0	5.30
Bub.Sort	49	11,028,902	66,560	865	13.53	66,560	865	0.32
Ins.Sort	49	2,619,985	15,054	68	2.98	15,054	68	0.39
AES [1]	789	534	171	39	0.28	171	39	1.43
DES [1]	368	580	108	11	0.29	108	11	0.17

3 APPLICATION SCENARIOS

We now demonstrate how CANAL may be used by KLEE, SMACK, and Crab-llvm using five example programs taken from the SV-COMP benchmark [2]: `copysome`, `sanfoundry`, and `standard` from the `array-programs` section, and `gcd` and `sum` from the `bit-vectors` 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 $P(k)$ with sensitive input k has timing side-channel leaks if the execution time of P depends on the value of k. That is, $\exists k_1, k_2 \in dom(k) : \tau(P, k_1) \neq \tau(P, k_2)$, where k_1 and k_2 are values in the domain of k and τ is the execution time. Even if the program executes the same number (and type) of instructions, the execution time may still differ if there are different numbers of cache hits/misses. Such side-channel leaks may be detected by CANAL + KLEE.

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

```
klee_make_symbolic(&input1);
klee_make_symbolic(&input2);
__CSIM_init_cache();
prog(input1);
h1 = __CSIM_num_hit;
m1 = __CSIM_num_miss;
__CSIM_init_cache();
prog(input2);
h2 = __CSIM_num_hit;
m2 = __CSIM_num_miss;
assert( h1==h2 && m1==m2 );
```

Figure 3: CANAL for timing side channel detection.

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.

		CANAL + KLEE			Time (s)		
Name	LoC	Detection result	Total-tests	Leaky-tests	1 KB	16 KB	32 KB
copysome	79	No leak	121	0	1.38	5.05	13.19
sanfoundry	95	No leak	81	0	1.32	2.16	2.82
standard	61	Leak	9	1	0.27	0.69	1.10
gcd	52	Leak	6	1	0.11	0.41	0.77
sum	56	Leak	6	1	0.07	0.38	0.74

3.2 Combined with Software Verification Tools

While symbolic execution is geared toward generating tests, verification tools such as SMACK are geared toward generating proofs, e.g., proving that an assertion holds under all test inputs. We show how SMACK can leverage CANAL to prove cache related properties.

Must-hits and Must-misses. One type of properties of interest is assertions over auxiliary variables such as `__CSIM_Load_ret` and `__CSIM_Store_ret`. For example, if a certain `Load` or `Store` instruction in the program always leads to a cache hit or miss, regardless of the program path and test input; in such a case, we call it a Must-hit or a Must-miss.

CANAL instrument the LLVM bit-code in such a way that calls to `__CSIM_Load()` and `__CSIM_Store()` set the values of auxiliary variables `__CSIM_Load_ret` and `__CSIM_Store_ret` to reflect the cache status: *true* means the memory access leads to a hit, whereas *false* means it leads to a miss.

Figure 4 shows a program where we check if read of `buffer[2]` is a Must-hit. Thus, we save the value of `__CSIM_Load_ret` immediately after the read of `buffer[2]` to the variable named `h` and add an assertion stating `h` should always be *true*. If SMACK can prove the assertion, we know the read of `buffer[2]` is a Must-hit. Alternatively, we can add `assert(h==false)` and use SMACK to prove it is a Must-miss.

Since program verification is undecided in general (e.g., equivalent to the Turing-halting problem), SMACK may fail to prove either assertion; in such a case, the result remains inconclusive. In this particular example, however, SMACK is able to find a violation of the Must-hit assertion and generate a counterexample. The counterexample shows a scenario where `buffer[0]` and `buffer[16]` resides in two different 64-byte cache lines.

Table 3 shows the results of applying CANAL+SMACK to assertions we manually inserted to check if a `Load` or `Store` instruction in the program is a Must-hit/miss. With loop-unrolling bound of SMACK set to 10, and the cache size set to 1 KB, SMACK successfully verified all assertions. However, when the cache size was increased to 16 KB and 32 KB, SMACK started to timeout on some programs. This points out a scalability limitation of SMACK, together with direction for future work: improving the verification algorithms to make SMACK (and similar tools) more scalable for non-functional properties.

```
if (cond)  buffer[0] = x;
else       buffer[16] = y;
z = buffer[2];
h = __CSIM_Load_ret;
assert(h == true); // `Must-Hit'?
```

Figure 4: CANAL for must-hit analysis.

Table 3: Results of the Must-hit analysis.

		CANAL + SMACK			Time (s)		
Name	LoC	Loop-unroll-bound	Property	Results	1 KB	16 KB	32 KB
copysome	69	10	Must-miss	Verified	226.87	TO	TO
sanfoundry	75	10	Must-miss	Verified	78.08	1055.55	TO
standard	38	10	Must-hit	Verified	24.53	139.59	344.71
gcd	74	10	Must-miss	Verified	22.88	142.78	375.27
sum	42	10	Must-miss	Verified	36.53	257.68	723.21

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 ($5 \leq$ *__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.

```
if (cond)  buffer[0] = 1;
else       buffer[16] = 1;
buffer[5] = 1;
n_s_h = __CSIM_num_Store_hit;
n_s_m = __CSIM_num_Store_miss;
n_s = n_s_h + n_s_m;
__CRAB_assert(n_s_h  > 1);
__CRAB_assert(n_s_m < 3);
__CRAB_assert(n_s == 2);
```

Figure 5: CANAL for computing value ranges.

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: Results of the numerical abstract interpretation.

		CANAL + Crab-llvm				Time (s)		
Name	LoC	S.Hit	S.Miss	L.Hit	L.Miss	1 KB	16 KB	32 KB
copysome	75	[1, ∞]	[2, ∞]	[1, ∞]	[0, ∞]	73.77	937.23	TO
sanfoundry	85	[0, ∞]	[3, ∞]	[4, ∞]	[1, ∞]	67.17	636.89	2416.82
standard	58	[0, ∞]	[1, ∞]	[1, ∞]	[0, 0]	13.06	528.53	2188.98
gcd	82	[0, ∞]	[2, ∞]	[6, ∞]	[0, ∞]	3.99	105.59	382.53
sum	54	[0, ∞]	[3, 3]	[2, ∞]	[0, 0]	0.87	39.03	146.01
copysome-unroll	105	[22, 22]	[4, 4]	[43, 43]	[0, 0]	91.62	452.34	1303.80
sanfoundry-unroll	168	[5, 15]	[3, 3]	[14, 29]	[1, 5]	19.04	296.64	1149.54
standard-unroll	130	[10, 19]	[2, 11]	[20, 20]	[0, 0]	21.11	279.30	985.24
gcd-unroll	107	[0, 13]	[2, 3]	[6, 27]	[0, 0]	11.69	174.16	608.06
sum-unroll	123	[6, 12]	[3, 3]	[20, 32]	[0, 0]	12.51	197.88	688.32

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 $+\infty$. 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 variable by analyzing the symbol table of the pre-compiled code. 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_Store()`. Although C code is used in Figure 6, this is actually implemented at the LLVM bitcode level inside CANAL.

```
int var;//its cache `set' and `tag'
        are 242 and 1, respectively
var = 2;
__CSIM_Store(242, 1);
```

Figure 6: Pre-computed 'set' and 'tag' values.

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[i]`. 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()`.

```
buffer[i] = 20;
if (address_of_buffer + 4*i <
       __CSIM_addr_of_cacheline01) {
    __CSIM_Store(242, 0);
} else if (address_of_test + 4*i <
       __CSIM_addr_of_cacheline02) {
      __CSIM_Store(242, 1);
} else {
      __CSIM_Store(242, 3);
}
```

Figure 7: Dynamically computed 'set' and 'tag' values.

Figure 8 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_cacheline00_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.

```
function __CSIM_Store(set, tag) {
  if (set == 0) {
    if (__CSIM_cacheline00_taken &&
        __CSIM_cacheline00_tag==tag){
      // cache hit
      __CSIM_num_Store_hit ++;
      __CSIM_Store_ret = true;
    } else if (...) {
      // cache hit
      ...
    } else {
      // cache miss
      __CSIM_num_Store_miss ++;
      __CSIM_Store_ret = false;
      // pick a new line based on
      the update policy
      ...
    }
  } else if (set == 1) {
    ...
  } else if (set == 2) {
    ...
  }
  ...
}
```

Figure 8: Code snippet of `__CSIM_Store`.

Implementations of functions `__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 precomputation 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. CANAL allows standard software verification tools to check a new class of cache timing related properties. We have 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.

REFERENCES

[1] AES/DES in openSSL. https://github.com/openssl/openssl/tree/OpenSSL_0_9_7-stable/crypto/. Accessed: 2018-04-18.
[2] Dirk Beyer. Software verification and verifiable witnesses. In *TACAS*, pages 401–416, 2015.
[3] M. Luisa Córdoba Cabeza, M. Isabel García Clemente, and M. Luz Rubio. CacheSim: A cache simulator for teaching memory hierarchy behaviour. In *SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, 1999.
[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.
[5] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *RTSS*, pages 193–203, 2011.
[6] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. Precise cache timing analysis via symbolic execution. In *RTAS*, pages 1–12, 2016.
[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
[8] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Security*, pages 431–446, 2013.
[9] Nathan Binkert et al. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.
[10] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald SØndergaard, and Peter J. Stuckey. An abstract domain of uninterpreted functions. In *VMCAI*, pages 85–103, 2016.
[11] Shengjian Guo, Meng Wu, and Chao Wang. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *FSE*, 2018.
[12] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce Jacob. CMP $im: A pin-based on-the-fly multi-core cache simulator. In *Workshop on Modeling, Benchmarking and Simulation, co-located with ISCA*, pages 28–36, 2008.
[13] Y-TS Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *RTSS*, pages 254–263, 1996.
[14] Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *CAV*, pages 106–113, 2014.
[15] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *ISSTA*, pages 15–26, 2018.