saarland-informatics-campus.de

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis

Valentin Touzeau Jan Reineke



This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 101020415)"



UNIVERSITÄT DES SAARLANDES

SIC Saarland Informatics Campus

Cache Analysis is Important



Cache Analysis is Important





Cache Analysis is Important





LOAD r2, _a LOAD r1, _b ADD r3, r2, r1









Source program





Binary program

0000000	0000	0001	0001	1010	0010	0001	0004	0128
0000010	0000	0016	0000	0028	0000	0010	0000	0020
0000020	0000	0001	0004	0000	0000	0000	0000	0000
0000030	0000	0000	0000	0010	0000	0000	0000	0204
0000040	0004	8384	0084	c7c8	00c8	4748	0048	e8e9
0000050	00e9	6a69	0069	a8a9	00a9	2828	0028	fdfc
0000060	00fc	1819	0019	9898	0098	d9d8	00d8	5857
0000070	0057	7b7a	007a	bab9	00b9	3a3c	003c	8888
0000080	8888	8888	8888	8888	288e	be88	8888	8888
0000090	3b83	5788	8888	8888	7667	778e	8828	8888
00000a0	d61f	7abd	8818	8888	467c	585f	8814	8188
00000b0	8b06	e8f7	88aa	8388	8b3b	88f3	88bd	e988
00000c0	8a18	880c	e841	c988	b328	6871	688e	958b
00000d0	a948	5862	5884	7e81	3788	1ab4	5a84	3eec
00000e0	3d86	dcb8	5cbb	8888	8888	8888	8888	8888
00000f0	8888	8888	8888	8888	8888	8888	8888	0000
0000100	0000	0000	0000	0000	0000	0000	0000	0000
*								
0000130	0000	0000	0000	0000	0000	0000	0000	
000013e								



Source program





Binary program

0000000	0000	0001	0001	1010	0010	0001	0004	0128
0000010	0000	0016	0000	0028	0000	0010	0000	0020
0000020	0000	0001	0004	0000	0000	0000	0000	0000
0000030	0000	0000	0000	0010	0000	0000	0000	0204
0000040	0004	8384	0084	c7c8	00c8	4748	0048	e8e9
0000050	00e9	6a69	0069	a8a9	00a9	2828	0028	fdfc
0000060	00fc	1819	0019	9898	0098	d9d8	00d8	5857
0000070	0057	7b7a	007a	bab9	00b9	3a3c	003c	8888
0000080	8888	8888	8888	8888	288e	be88	8888	8888
0000090	3b83	5788	8888	8888	7667	778e	8828	8888
00000a0	d61f	7abd	8818	8888	467c	585f	8814	8188
00000b0	8b06	e8f7	88aa	8388	8b3b	88f3	88bd	e988
00000c0	8a18	880c	e841	c988	b328	6871	688e	958b
00000d0	a948	5862	5884	7e81	3788	1ab4	5a84	Зеес
00000e0	3d86	dcb8	5cbb	8888	8888	8888	8888	8888
00000f0	8888	8888	8888	8888	8888	8888	8888	0000
0000100	0000	0000	0000	0000	0000	0000	0000	0000
*								
0000130	0000	0000	0000	0000	0000	0000	0000	
000013e								









Source program





Binary program

0000000	0000	0001	0001	1010	0010	0001	0004	0128
0000010	0000	0016	0000	0028	0000	0010	0000	0020
0000020	0000	0001	0004	0000	0000	0000	0000	0000
0000030	0000	0000	0000	0010	0000	0000	0000	0204
0000040	0004	8384	0084	c7c8	00c8	4748	0048	e8e9
0000050	00e9	6a69	0069	a8a9	00a9	2828	0028	fdfc
0000060	00fc	1819	0019	9898	0098	d9d8	00d8	5857
0000070	0057	7b7a	007a	bab9	00b9	3a3c	003c	8888
0000080	8888	8888	8888	8888	288e	be88	8888	8888
0000090	3b83	5788	8888	8888	7667	778e	8828	8888
00000a0	d61f	7abd	8818	8888	467c	585f	8814	8188
00000b0	8b06	e8f7	88aa	8388	8b3b	88f3	88bd	e988
00000c0	8a18	880c	e841	c988	b328	6871	688e	958b
00000d0	a948	5862	5884	7e81	3788	1ab4	5a84	Зеес
00000e0	3d86	dcb8	5cbb	8888	8888	8888	8888	8888
00000f0	8888	8888	8888	8888	8888	8888	8888	0000
0000100	0000	0000	0000	0000	0000	0000	0000	0000
*								
0000130	0000	0000	0000	0000	0000	0000	0000	
000013e								





а

С

Control-flow graph



Instruction Cache Analysis

Classification: "always hit" "always miss" "unknown"



Source program





4













First Contribution: Symbolic Control-Flow Graphs





First Contribution:





First Contribution:





Captures dependence of addresses on loop iteration!





Loop variables capture iteration counts, here *i* and *j*.





- Loop variables capture iteration counts, here *i* and *j*.
- Three ways to manipulate variables:





- Loop variables capture iteration counts, here *i* and *j*.
- Three ways to manipulate variables:
 - $entry_i$ reset variable *i* to 0





- Loop variables capture iteration counts, here *i* and *j*.
- Three ways to manipulate variables:
 - $entry_i$ reset variable *i* to 0
- $backedge_i$ increment variable i





- Loop variables capture iteration counts, here *i* and *j*.
- Three ways to manipulate variables:
 - $entry_i$ reset variable *i* to 0
- $backedge_i$ increment variable i
- can only take edge if variable *i* $assume_{i,e}$ is equal to expression *e*





- Loop variables capture iteration counts, here *i* and *j*.
- Three ways to manipulate variables:
 - $entry_i$ reset variable *i* to 0
- $backedge_i$ increment variable i
- $assume_{i,e}$ can only take edge if variable iis equal to expression *e*
- Addresses of memory accesses captured as polynomial expressions of loop variables.





Obtained from LLVM's ScalarEvolution Analysis Pass

- Loop variables capture iteration counts, here *i* and *j*.
- Three ways to manipulate variables:
 - $entry_i$ reset variable *i* to 0
- $backedge_i$ increment variable i
- $assume_{i,e}$ can only take edge if variable iis equal to expression *e*
- Addresses of memory accesses captured as polynomial expressions of loop variables.











First loop: i = 0, A[0] i = 1, A[1]



First loop:i = 0, A[0]i = 1, A[1]...i = 99, A[99]



First loop:i = 0, A[0]i = 1, A[1]...i = 99, A[99]

Second loop:



First loop: i = 0, A[0] i = 1, A[1] ... i = 99, A[99]

Second loop: j = 0, A[99]



First loop: i = 0, A[0] i = 1, A[1] ... i = 99, A[99]

Second loop: j = 0, A[99] j = 1, A[98]



First loop: i = 0, A[0] i = 1, A[1] ... i = 99, A[99]

Second loop: j = 0, A[99] j = 1, A[98] ... j = 99, A[0]



Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively



Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Cache Analysis: Intuitively

i = 0, A[0]

i = 1, A[1] i = 2, A[2] i = 3, A[3]







i = 0, A[0]

Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line





Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line









Cache Analysis: Intuitively i = 2, A[2]i = 1, A[1]i = 3, A[3]

8




Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line









Cache Analysis: Intuitively



i = 1, A[1]i = 2, A[2]i = 3, A[3]





Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line















Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line















Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line















Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line









Cache Analysis: Intuitively



i = 1, A[1]

A[0]



i = 2, A[2]

i = 3, A[3]

8





Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line









Cache Analysis: Intuitively





i = 2, A[2]**A**[2] **A**[0]

i = 3, A[3]





Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line









Cache Analysis: Intuitively



i = 1, A[1]





i = 2, A[2]









Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line















Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line















Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line









Cache Analysis: Intuitively



i = 1, A[1]





i = 2, A[2]





A[2] **A**[0]

8





Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line



j = 0, A[99]

j = 1, A[98]j = 98, A[1]j = 99, A[0]• • •





Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line



Cache Analysis: Intuitively









Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line







Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line



j = 99, A[0]





Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line



Cache Analysis: Intuitively





8





Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line



Cache Analysis: Intuitively



A[98]

A[96]







Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line







Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line







Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line



Cache Analysis: Intuitively



A[98]

A[96]

A[98]

A[96]

$$j = 1, A[98] \dots j = 98, A[1] j = 99, A[0]$$





Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line



Cache Analysis: Intuitively



• • •



A[98]

A[96]



i = 99, A[0]





Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line







Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line







Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line







Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line







Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line







Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Challenges in Data Cache Analysis II



Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Challenges in Data Cache Analysis II

1. Cache states depend on loop iteration



Challenges in Data Cache Analysis II 1. Cache states depend on loop iteration Contribution: **Symbolic Cache States**

Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line



Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Challenges in Data Cache Analysis II 1. Cache states depend on loop iteration Contribution: Symbolic Cache States

2. Behavior is phase dependent: Warm-up phase: hits/misses depending on initial state Steady-state phase: repetitive patterns



Assumptions:

- fully-associative cache
- associativity 2
- least-recently-used
- 2 array cells per cache line

Challenges in Data Cache Analysis II 1. Cache states depend on loop iteration Contribution: Symbolic Cache States

2. Behavior is phase dependent: Warm-up phase: hits/misses depending on initial state Steady-state phase: repetitive patterns

Contribution: Context-sensitive Analysis









First loop:





68

First loop: i = 0, A[i] i = 1, A[i] i = 2, A[i]





i = 3, A[i] ... i = 100

69



First loop: i = 0, A[i] i = 1, A[i]





i = 2, A[i] i = 3, A[i] i = 100• • •





First loop: i = 0, A[i]



i = 1, A[i]





i = 2, A[i] i = 3, A[i] ... i = 100



71



First loop: i = 0, A[i]





i = 1, A[i]





i = 2, A[i] i = 3, A[i] i = 100• • •






First loop: i = 0, A[i]





i = 1, A[i]





i = 2, A[i] Cache Hit Spatial Locality

i = 3, A[i]i = 100• • •











i = 2, A[i] Cache Hit Spatial Locality

i = 3, A[i]i = 100• • •











i = 2, A[i] i = 3, A[i] ... i = 100











i = 2, A[i] i = 3, A[i] i = 100• • •













i = 3, A[i]i = 100• • •













A[i]

A[i-2]



i = 3, A[i]

i = 100

• • •















i = 3, A[i]

i = 100• • •

Cache Hit Spatial Locality



















A[i-2]























• • •



i = 100





Second loop:













• • •



i = 100





Second loop: j = 0, A[99-j] = 1, A[99-j]



• • •











• • •

Second loop: j = 0, A[99-j] = 1, A[99-j]











Second loop: j = 0, A[99-j] = 1, A[99-j]



• • •











Second loop: j = 0, A[99-j] = 1, A[99-j]



• • •











Second loop: j = 0, A[99-j] = 1, A[99-j]





• • •













Second loop: j = 0, A[99-j] = 1, A[99-j]





• • •













• • •

































• • •















• • •







i = 100













j = 98, A[99-j] j = 99, A[99-j]





• • •







i = 100



SIC Saarland Informatics Campus











j = 98, A[99-j] j = 99, A[99-j]







• • •









i = 100





SIC Saarland Informatics Campus



































j = 98, A[99-j] j = 99, A[99-j]









• • •









i = 100

SIC Saarland Informatics Campus



























- influence analysis accuracy + cost

• chosen heuristically based on cache geometry + loop structure



Does it work?



Does it work?

Accuracy: Does symbolic analysis improve bounds on cache misses?





Does it work?

Accuracy: Does symbolic analysis improve bounds on cache misses?

Scalability: How does symbolic analysis runtime scale with program loop bounds?













#old misses #new misses




#old misses #new misses

time limit = 1 hour



13



#old misses #new misses

time limit = 1 hour

PolyBench = Polyhedral Benchmark Suite





Accuracy

#old misses #new misses

> time limit = 1 hour



PolyBench = Polyhedral Benchmark Suite

13

Saarland Informatics Campus









Data size XS to XL

Analysis time (seconds)

PolyBench = Polyhedral Benchmark Suite





Scalability

Data size XS to XL

Analysis time (seconds)



PolyBench = Polyhedral Benchmark Suite

Saarland Informatics Campus







Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis Saarland University Saarland Informatics Campus

Saarbrücken, Germany reineke@cs.uni-saarland.de

Cache analysis aims to statically characterize a program's

1) A transformation of the program under analysis into a simpler program abstraction: a control-flow graph (CFG)

whose edges are decorated with memory accesses.

as "always hit", "always miss", or "unknown".

for (int x = 0; x < 100; x++)
sum += A[x]</pre>

and so the corresponding edge in the CFG needs to be

and so the corresponding edge in the CrO needs to be conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes in which the array elements are accessed is lost and it becomes impossible to make accurate predictions about the program's

accurate predictions about the program s abstraction that more precisely to the behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed. captures a program s memory access benavior is thus needed. Our first contribution is the definition of symbolic controlflow graphs in Section which is our formalization of the

a manner that is amenable to static analysis.

main memory can take hundreds of cycles. This variability is a challenge in the context of real-time systems, where it is necessary to bound a program's worst-case classical LRU must analysis [10], [11] from plain to symbolic data cache

This variability is a challenge in the context of real-time systems, where it is necessary to bound a program's worst-case execution time (WCET) [1] to guarantee that safety-critical control-flow graphs. To fully realize the potential of symbolic

systems, where it is necessary to bound a program's worst-case execution time (WCET) [II] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET

execution time (WCET) [II] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The analysis combining loop peeling and unrolling in Section [VI]

analysis it is thus imperative to take caches into account. The analysis combining loop peeling and unrolling in analysis variability induced by caches also introduces security challenges. Implementations of cryptographic algorithms have

applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The analysis variability induced by caches also introduces security and various implementation tricks in Section VII.

challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], ④, ⑤ may help to uncover such been shown to be vulnerable to cache timing attacks [2] suite in Section VIII demonstrates that symbolic cache analysis and cache analysis [3], [4], [5] may help to uncover such terms of accuracy and analysis runtime.

now graphs in Section W which is our iormalization of the output of LLVM's ScalarEvolution analysis [6], [2]. Symbolic

output of LLVM's ScalarEvolution analysis [Q], [L]. Symbolic CFGs accurately capture the link between loop iterations and

Cros accurately capture the link between 100p iterations and accessed memory blocks via chains of recurrences [8], [9] in

To exploit this more expressive program abstraction our

main contribution is the development of symbolic data cache

Valentin Touzeau Saarland University Saarland Informatics Campus Saarbrücken, Germany valentin.touzeau@cs.uni-saarland.de

cache behavior by classifying memory accesses in the program

cache benavior by classifying memory accesses in me program as guaranteed cache hits or misses. One perspective on cache as guaranteed cache mus or misses. One perspective analysis is that it is the composition of two phases: Abstract—While instruction cache analysis is essentially lead methods data cache malurie is more challenging to Abstract—While instruction cache analysis is essentially solved problem, data cache analysis is more challenging. I

1) A transformation of the program under analysis into a solved problem, data cache analysis is more challenging. contrast to instruction fetches, the data accesses generated i contrast to instruction fetches, the data accesses generated by a memory instruction may vary with the program's inputs and a memory instruction may vary with the program's inputs an across dynamic occurrences of the same instruction in loops. across dynamic occurrences of the same instruction in loops. We observe that the plain control-flow graph (CFG) straction employed in classical cache analyses is inadequate to conture the dynamic behavior of memory instructions. On top of

whose edges are decorated with memory accesses. 2) An analysis of this decorated CFG that classifies accesses as always mt, always miss, or unknown. For instruction cache analysis this two-phase approach works ror instruction cache analysis uns two-phase approach works well, as CFGs accurately captures most programs' instruction straction employed in classical cache analyses is inadequate to capture the dynamic behavior of memory instructions. On top of plain CFGs, accurate analysis of the underlying program's cache behavior is impossible. behavior is impossible. Thus, our first contribution is the definition of a more

wen, as CrOs accurately captures most programs instruction fetch sequences. For data cache analysis, however, a plain CFG abstraction can be highly inaccurate. Consider for example the Thus, our first contribution is the definition of a more expressive program abstraction coined symbolic control-flow graphs, which can be obtained from LLVM's ScalarEvolution analysis. To exploit this richer abstraction our main contribution

graphs, which can be obtained from LLVM's ScalarEvolution analysis. To exploit this richer abstraction, our main contribution is the development of symbolic data cache analysis, a smooth generalization of classical LRU must analysis from plain to symbolic control-flow graphs. following simple loop: generalization of classical LRU must analysis from plain to symbolic control-flow graphs. The experimental evaluation demonstrates that symbolic data cache analysis consistently outperforms classical LRU must analysis both in terms of accuracy and analysis runtime.

cache analysis consistently outperforms classical LRU analysis both in terms of accuracy and analysis runtime. Index Terms—cache analysis, chains of recurrences, u webes symbolic analysis

In each iteration of the loop a different address is accessed,

between the processor cores and main memory.

to DRAM-based main memory is much higher than the latency

of arithmetic and logic computations on processor cores. This

vulnerabilities or prove their absence.

of arithmetic and logic computations on processor cores. This "memory gap" is commonly tackled by a hierarchy of caches

In the presence of caches, the latency of a memory access

In the presence of caches, the fatency of a memory access may vary widely depending on the level of the memory

may vary widely depending on the level of the memory hierarchy that is able to serve the access. Hits to the first-

level cache take just a few processor cycles, while accesses that miss in all cache levels and thus need to be accessed to be

that miss in all cache levels and thus need to be served by

caches, symbolic analysis Due to technological developments, the latency of accesses



Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis Saarland University Saarland Informatics Campus

Saarbrücken, Germany reineke@cs.uni-saarland.de

following simple loop:

This variability is a challenge in the context of real-time systems, where it is necessary to bound a program's worst-case execution time (WCET) [1] to guarantee that safety-critical control-flow graphs. To fully realize the potential of symbolic

systems, where it is necessary to bound a program's worst-case execution time (WCET) [II] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET is necessary with the program's worst-case exercises are used to be accurate with the program's worst-case interval analysis (III), III) to guarantee that safety-critical data cache analysis we further introduce a context-sensitive data cache analysis we further introduce d

execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The analysis combining loop peeling and unrolling in Section [1]

analysis it is thus imperative to take caches into account. The analysis combining loop peeling and unrolling in analysis variability induced by caches also introduces security challenges. Implementations of cryptographic algorithms have

analysis it is thus imperative to take caches also introduces security and various implementation tricks in Section VII.

challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], ④, ⑤] may help to uncover such been shown to be vulnerable to cache timing attacks [2] suite in Section VIII demonstrates that symbolic cache analysis and cache analysis [3], [4], [5] may help to uncover such vulnerabilities or prove their absence.

Cache analysis aims to statically characterize a program's

1) A transformation of the program under analysis into a () A transformation of the program under analysis muo a simpler program abstraction: a control-flow graph (CFG)

whose edges are decorated with memory accesses.

as "always hit", "always miss", or "unknown".

for (int x = 0; x < 100; x++)
 sum += A[x]</pre>

In each iteration of the loop a different address is accessed,

and so the corresponding edge in the CFG needs to be

and so the corresponding edge in the CPU needs to be conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes

in which the array elements are accessed is lost and it becomes impossible to make accurate predictions about the program's

cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed. Captures a program s memory access benavior is unus needed. Our first contribution is the definition of symbolic controlflow graphs in Section which is our formalization of the

a manner that is amenable to static analysis.

now graphs in Section W which is our rormanzation or the output of LLVM's ScalarEvolution analysis [6], [7]. Symbolic

output of LLVM's ScalarEvolution analysis [0], [1]. Symbolic CFGs accurately capture the link between loop iterations and

accessed memory blocks via chains of recurrences [8], [9] in

To exploit this more expressive program abstraction our

main contribution is the development of symbolic data cache

analysis in Section 1 a smooth generalization of Ferdinand's classical I PU must analysis (TTP) (TTP) from plain to compare

2) An analysis of this decorated CFG that classifies accesses

Valentin Touzeau Saarland University Saarland Informatics Campus Saarbrücken, Germany valentin.touzeau@cs.uni-saarland.de

cache anarysis anns to staticary characterize a program s cache behavior by classifying memory accesses in the program

as guaranteed cache hits or misses. One perspective on cache as guaranteeu cache nus or nusses. One perspective analysis is that it is the composition of two phases: problem, data cache analysis is more challenging. analysis is u provident, under cache allarysis is more chanceligned ast to instruction fetches, the data accesses generated o instruction retches, the usua accesses Benerated by instruction may vary with the program's inputs and occurrences of the same instruction in loops.

that the plain control-flow graph (CFG) abat the plant control-now graph (CrO) and the classical cache analyses is inadequate to straction employed in classical cache analyses is madequate to capture the dynamic behavior of memory instructions. On top of rabin CFCe, compute analysis of the underlying program's cache

vapture the dynamic behavior of memory instructions. On top of lain CFGs, accurate analysis of the underlying program's cache first contribution is the definition of a more

as always int, always miss, or unsubwit. For instruction cache analysis this two-phase approach works ror instruction cache analysis uns two-phase approach works well, as CFGs accurately captures most programs' instruction austraction comen symposic control-now e obtained from LLVM's ScalarEvolution havior is im

fetch sequences. For data cache analysis, however, a plain CFG abstraction can be highly inaccurate. Consider for example the program abstraction coined \$ is, To exploit this richer abstraction, our main contribution t of symbolic data cache analysis, a smooth Thus, our

the development of symbolic data cache analysis, a smoo neralization of classical LRU must analysis from plain t The experimental evaluation demonstrates that symbolic data

ue experimental evaluation demonstrates that symbolic analysis consistently outperforms classical LRU is both in terms of accuracy and evaluation of the symbols of the sym is the develop symbolic control-flow graphs.

cache analysis consistently outperforms classical LRU analysis both in terms of accuracy and analysis runtime. Index Terms—cache analysis, chains of recurrences,

between the processor cores and main memory.

to DRAM-based main memory is much higher than the latency

of arithmetic and logic computations on processor cores. This

vulnerabilities or prove their absence.

or anumeuc and logic computations on processor cores. This "memory gap" is commonly tackled by a hierarchy of caches

In the presence of caches, the latency of a memory access

in the presence of caches, the fatency of a memory access may vary widely depending on the level of the memory

hierarchy that is able to serve the access. Hits to the first-

level cache take just a few processor cycles, while accesses that miss in all cache levels and thus need to be

level cache take just a rew processor cycles, while accesses that miss in all cache levels and thus need to be served by

caches, symbolic analysis Due to technological developments, the latency of accesses

S C Saarland Informatics Campus

Formalization as Abstract Interpretation

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis Saarland Informatics Campus

Saarbrücken, Germany reineke@cs.uni-saarland.de

following simple loop:

systems, where it is necessary to bound a program's worst-case execution time (WCET) [II] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET

execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The analysis combining loop peeling and unrolling in Section ∇

challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks 2 and cache analysis 3, 4, 5 may help to uncover such been shown to be vulnerable to cache timing attacks [2] suite in Section VIII demonstrates that symbolic cache analysis and cache analysis [3], [4], [5] may help to uncover such vulnerabilities or prove their absence.

Cache analysis aims to statically characterize a program'.

simpler program abstraction: a control-flow graph (CFG) whose edges are decorated with memory accesses. Whose edges are decorated with memory accesses. An analysis of this decorated CFG that classifies accesses

as always int, always integers, or unknown instruction cache analysis this two-phase approach works

In each iteration of the loop a different address is accessed, and so the corresponding edge in the CFG needs to be

conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes in which the array elements are accessed is tost and it occornes impossible to make accurate predictions about the program's

cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed.

a manner that is amenable to static analysis.

Our first contribution is the definition of symbolic controlflow graphs in Section which is our formalization of the

output of LLVM's ScalarEvolution analysis 6, [7]. Symbolic

Output of LLV N S ScatarEvolution analysis [0], [14]. Symbolic CFGs accurately capture the link between loop iterations and

accessed memory blocks via chains of recurrences [8], [9] in

To exploit this more expressive program abstraction our

main contribution is the development of symbolic data cache

analysis in Section A a smooth generalization of Ferdinand's

classical LRU must analysis IID, III from plain to symbolic

The experimental evaluation on the PolyBench benchmark

as "always hit", "always miss", or "unknown".

for (int x = 0; x < 100; x++)
 sum += A[x]</pre>

Valentin Touzeau Saarland University Saarland Informatics Campus Saarbrücken, Germany valentin.touzeau@cs.uni-saarland.de

cache behavior by classifying memory accesses in the program as guaranteed cache hits or misses. One perspective on cache as guaranteed cache nits or misses. One perspective analysis is that it is the composition of two phases: data cache analysis i fetches, the data accea

1) A transformation of the program under analysis into a vary with the program the same instruction in loops. graph (CFG) abcontrol-flow f memory instructions. On top of al cache analyses is

well, as CFGs accurately captures most programs' instruction fetch sequences. For data cache analysis, however, a plain CFG

abstraction can be highly inaccurate. Consider for example the plic data cache LRU must analysis

classical

cache analysis consistentity outperforms classical LK analysis both in terms of accuracy and analysis runtime. Index Terms cooke analysis abains of security

to DRAM-based main memory is much higher than the latency

of arithmetic and logic computations on processor cores. This

in anumence and logic computations on processor cores. This "memory gap" is commonly tackled by a hierarchy of caches

vulnerabilities or prove their absence.

In the presence of caches, the latency of a memory access

level cache take just a few processor cycles, while accesses

that miss in all cache levels and thus need to be served by

y that is able to serve the access. Hits to the first-

applications meet all of their deautiles. For accurate wCE1 analysis it is thus imperative to take caches into account. The

in the presence of caches, the fatency of a memory access may vary widely depending on the level of the memory

Terms—cache analysis, chains

between the processor cores and main memory.

caches, symbolic analysis Due to technological developments, the latency of accesses

C Saarland Informatics Campus

 Formalization as Abstract Interpretation • Multivariate chains of recurrences to represent symbolic expressions

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis Saarland Informatics Campus

Saarbrücken, Germany reineke@cs.uni-saarland.de

Cache analysis aims to statically characterize a program's

cache behavior by classifying memory accesses in the program

simpler program abstraction: a control-flow graph (CFG) whose edges are decorated with memory accesses. An analysis of this decorated CFG that classifies accesses

instruction cache analysis this two-phase approach works

In each iteration of the loop a different address is accessed, and so the corresponding edge in the CFG needs to be

conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes

in which the array elements are accessed is tost and it occornes impossible to make accurate predictions about the program's

cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed.

a manner that is amenable to static analysis.

Our first contribution is the definition of symbolic controlflow graphs in Section W which is our formalization of the

now graphs in Section Williams our roumanization of the output of LLVM's ScalarEvolution analysis [6], [7]. Symbolic

Output of LLVM'S ScalarEvolution analysis [0], [1]. Sympolic CFGs accurately capture the link between loop iterations and

accessed memory blocks via chains of recurrences [8], [9] in

To exploit this more expressive program abstraction our

main contribution is the development of symbolic data cache

analysis in Section A a smooth generalization of Ferdinand's

classical LRU must analysis IID, III from plain to symbolic

classical LEC must analysis [110], [111] from plan to symbolic control-flow graphs. To fully realize the potential of symbolic

analysis combining loop peeling and unrolling in Section V

The experimental evaluation on the PolyBench benchmark

as "always hit", "always miss", or "unknown".

for (int x = 0; x < 100; x++)

following simple loop:

execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The analysis combining loop beeling and unrolling in Section ∇

challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], [4], [5] may help to uncover such been shown to be vulnerable to cache timing attacks [2] suite in Section VIII demonstrates that symbolic cache analysis and cache analysis [3], [4], [5] may help to uncover such vulnerabilities or prove their absence.

Valentin Touzeau Saarland University Saarland Informatics Campus Saarbrücken, Germany valentin.touzeau@cs.uni-saarland.de

as guaranteed cache hits or misses. One perspective on cache as guaranced eache mis or misses. One perspective analysis is that it is the composition of two phases: with the program cache fetches, the aph (CFG) ab-

1) A transformation of the program under analysis (rec) v instructions. On top of

well, as CFGs accurately captures most programs' instruction fetch sequences. For data cache analysis, however, a plain CFG

abstraction can be highly inaccurate. Consider for example the data cache TRU

classical

's both in terms of accuracy and analysis runtim

between the processor cores and main memory.

to DRAM-based main memory is much higher than the latency

of arithmetic and logic computations on processor cores. This

vulnerabilities or prove their absence.

In the presence of caches, the latency of a memory access

level cache take just a few processor cycles, while accesses

that miss in all cache levels and thus need to be served by

presence of caches, the fatency of a memory access widely depending on the level of the memory

that is able to serve the access. Hits to the first-

applications meet all of their deadlines. For accurate wCE1 analysis it is thus imperative to take caches into account. The

mbolic analysis Due to technological developments, the latency of accesses

Terms-cache analysis,

C Saarland Informatics Campus

 Formalization as Abstract Interpretation • Multivariate chains of recurrences to represent symbolic expressions Implementation on top of LLVM

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis Saarland Informatics Campus

Saarbrücken, Germany

following simple loop:

execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The analysis combining loop peeling and unrolling in Section [V]

challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], [4], [5] may help to uncover such been shown to be vulnerable to cache timing attacks [2] suite in Section VIII demonstrates that symbolic cache analysis and cache analysis [3], [4], [5] may help to uncover such vulnerabilities or prove their absence.

reineke@cs.uni-saarland.de

Cache analysis aims to statically characterize a program

1) A transformation of the program under analysis into a simpler program abstraction: a control-flow graph (CFG) whose edges are decorated with memory accesses. An analysis of this decorated CFG that classifies accesses

as "always hit", "always miss", or "unknown".

instruction cache analysis this two-phase approach works

= 0; x < 100; x++)

In each iteration of the loop a different address is accessed, and so the corresponding edge in the CFG needs to be conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes

in which the array elements are accessed is lost and it occorne, impossible to make accurate predictions about the program's

cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed. Our first contribution is the definition of symbolic control flow graphs in Section W which is our formalization of the

output of LLVM's ScalarEvolution analysis [6], [7]. Symbolic

output of LLV N 5 ScalarEvolution analysis [0], [4]. Symbolic CFGs accurately capture the link between loop iterations and

accessed memory blocks via chains of recurrences [3], [9] is

a manner that is amenable to static analysis.

To exploit this more expressive program abstraction out

main contribution is the development of symbolic data cache

analysis in Section ∇ a smooth generalization of Ferdinand's

classical LRU must analysis IID, III from plain to symbolic

ciassical LRU must analysis [110], [110] from plan to symbolic control-flow graphs. To fully realize the potential of symbolic

analysis combining loop peeling and unrolling in Section V

The experimental evaluation on the PolyBench benchmark

Valentin Touzeau Saarland University Saarland Informatics Campus Saarbrücken, Germany valentin.touzeau@cs.uni-saarland.de

cache behavior by classifying memory accesses in the program

acue ochavior by classifying memory accesses in me program as guaranteed cache hits or misses. One perspective on cache analysis is that it is the composition of two phases: cache with the program vh (CFG) ab-

uctions. On top of

well, as CFGs accurately captures most programs' instruction

fetch sequences. For data cache analysis, however, a plain CFG abstraction can be highly inaccurate. Consider for example the

both in terms of accuracy and analysis runtim

to DRAM-based main memory is much higher than the latency

of arithmetic and logic computations on processor cores. This

vulnerabilities or prove their absence.

e presence of caches, the latency of a memory access

level cache take just a few processor cycles, while accesses

that miss in all cache levels and thus need to be served by

applications meet all of their deadlines. For accurate wCE1 analysis it is thus imperative to take caches into account. The

depending on the level of the memory

is able to serve the access. Hits to the first-

"memory gap'

between the pl

Due to technological developments, the latency of accesses polic analysis

Saarland Informatics

 Formalization as Abstract Interpretation • Multivariate chains of recurrences to represent symbolic expressions Implementation on top of LLVM • Discussion of related work

Leveraging LLVM's ScalarEvolution for Symbolic Data Cache Analysis Saarland Informatics Campus

Saarbrücken, Germany

following simple loop:

execution time (WCET) [1] to guarantee that safety-critical applications meet all of their deadlines. For accurate WCET analysis it is thus imperative to take caches into account. The analysis combining loop peeling and unrolling in Section [V]

challenges. Implementations of cryptographic algorithms have been shown to be vulnerable to cache timing attacks [2] and cache analysis [3], ④, ⑤) may help to uncover such been shown to be vulnerable to cache timing attacks [2] suite in Section VIII demonstrates that symbolic cache analysis and cache analysis [3], [4], [5] may help to uncover such terms of accuracy and analysis runtime.

reineke@cs.uni-saarland.de

Cache analysis aims to statically characterize a program

cache behavior by classifying memory accesses in the program

1) A transformation of the program under analysis into a simpler program abstraction: a control-flow graph (CFG) whose edges are decorated with memory accesses. An analysis of this decorated CFG that classifies accesses

as "always hit", "always miss", or "unknown".

instruction cache analysis this two-phase approach works

= 0; x < 100; x++)

In each iteration of the loop a different address is accessed, and so the corresponding edge in the CFG needs to be conservatively decorated with all possible addresses. The order in which the array elements are accessed is lost and it becomes

cache behavior. A program abstraction that more precisely captures a program's memory access behavior is thus needed. Our first contribution is the definition of symbolic control flow graphs in Section IV which is our formalization of the

output of LLVM's ScalarEvolution analysis [6], [7]. Symbolic

OUTPUT OF LLVIN S SCHAFEVOLUTION ANALYSIS [10], 12], 59110011C CFGs accurately capture the link between loop iterations and

accessed memory blocks via chains of recurrences [3], [9] i

To exploit this more expressive program abstraction

analysis in Section \mathbb{N} , a smooth generalization of Ferdinand's classical LRU must analysis IID, III from plain to symbolic

classical LICO must analysis [100], [110] from plan to symbolic control-flow graphs. To fully realize the potential of symbolic

analysis combining loop peeling and unrolling in Section V

The experimental evaluation on the PolyBench benchmark

contribution is the development of symbolic data cache

a manner that is amenable to static analysis.

captures most programs' instruction

make accurate predictions about the program's

Valentin Touzeau Saarland University Saarland Informatics Campus Saarbrücken, Germany valentin.touzeau@cs.uni-saarland.de

as guaranteed cache hits or misses. One perspective on cache analysis is that it is the composition of two phases:

fetch sequences. For data cache analysis, however, a plain CFG abstraction can be highly inaccurate. Consider for example the

both in terms of accuracy and analysis run

to DRAM-based main memory is much higher than the latency

of arithmetic and logic computations on processor cores. This

vulnerabilities or prove their absence.

nce of caches, the latency of a memory access

applications meet all of their deadlines. For accurate wCE1 analysis it is thus imperative to take caches into account. The

depending on the level of the memory

Cache levels and thus need to be served by

processor cycles, while accesses

Due to technological developments, the latency of accesses olic analysis

C Saarland Informatics Campus

 Formalization as Abstract Interpretation • Multivariate chains of recurrences to represent symbolic expressions Implementation on top of LLVM • Discussion of related work

Ouestions?