High-Precision Sound Analysis to Find Safety and Cybersecurity Defects

Daniel Kästner, Laurent Mauborgne, Stephan Wilhelm, Christian Ferdinand

AbsInt GmbH, 2020

Functional Safety

- Demonstration of functional correctness
  - Well-defined criteria
  - ➢ Automated and/or model-based testing
  - ➢ Formal techniques: model checking, theorem proving

Required by DO-178B / DO-178C / ISO-26262, EN-50128, IEC-61508

- Satisfaction of safety-relevant non-functional requirements
  - No runtime errors (e.g. division by zero, overflow, invalid pointer access, out-of-bounds array access)
  - Resource usage:
    - Timing requirements (e.g. WCET, WCRT)
    - Memory requirements (e.g. no stack overflow)
  - Robustness / freedom of interference (e.g. no corruption of content, incorrect synchronization, illegal read/write accesses)
  - ➢ Insufficient: Tests & Measurements
    - No specific test cases, unclear test end criteria, no full coverage possible
    - "Testing, in general, cannot show the absence of errors." [DO-178B]
  - Formal technique: abstract interpretation.

+ Security-relevant!

Required by DO-178B / DO-178C / ISO-26262, EN-50128, IEC-61508

(Information-/Cyber-) Security Aspects

- Confidentiality
  - Information shall not be disclosed to unauthorized entities
  - ⇨ safety-relevant
- Integrity
  - Data shall not be modified in an unauthorized or undetected way
  - ⇨ safety-relevant
- Availability
  - Data is accessible and usable upon demand
  - ⇨ safety-relevant

\+ Safety

In some cases: not safe $\Rightarrow$ not secure

In some cases: not secure $\Rightarrow$ not safe

Static Program Analysis

- General Definition: results only computed from program structure, without executing the program under analysis.
- Categories, depending on analysis depth:
  - Syntax-based: Coding guideline checkers (e.g. MISRA C)
  - Semantics-based

Question: Is there an error in the program?

- False positive: answer wrongly "Yes"
- False negative: answer wrongly "No" ☠

- Unsound: Bug-finders / bug-hunters.
  - False positives: possible
  - False negatives: possible
- Sound / Abstract Interpretation-based — Example: Astrée
  - False positives: possible
    Important: low false alarm rate
  - No false negatives ⇨ Soundness
    No defect missed

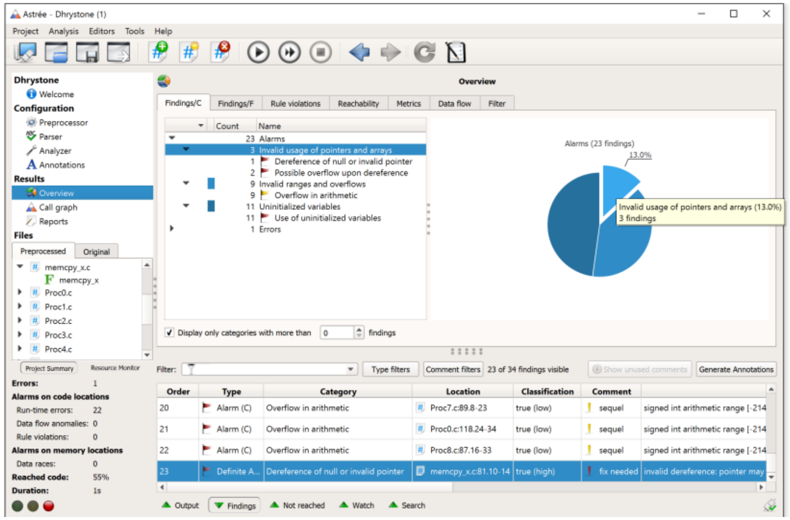
Support for Cybersecurity Analysis

- Many security vulnerabilities due to undefined / unspecified behaviors in the programming language semantics:
  - buffer overflows, invalid pointer accesses, uninitialized memory accesses, data races, etc.
  - Consequences: denial-of-service / code injection / data breach
- In addition:
  - Checking coding guidelines
  - Data and Control Flow Analysis
  - Impact analysis (data safety / “fault” propagation)
    - Program slicing
    - Taint analysis
  - Side channel attacks
    - SPECTRE detection (Spectre V1/V1.1, SplitSpectre)
  - …

AbsInt

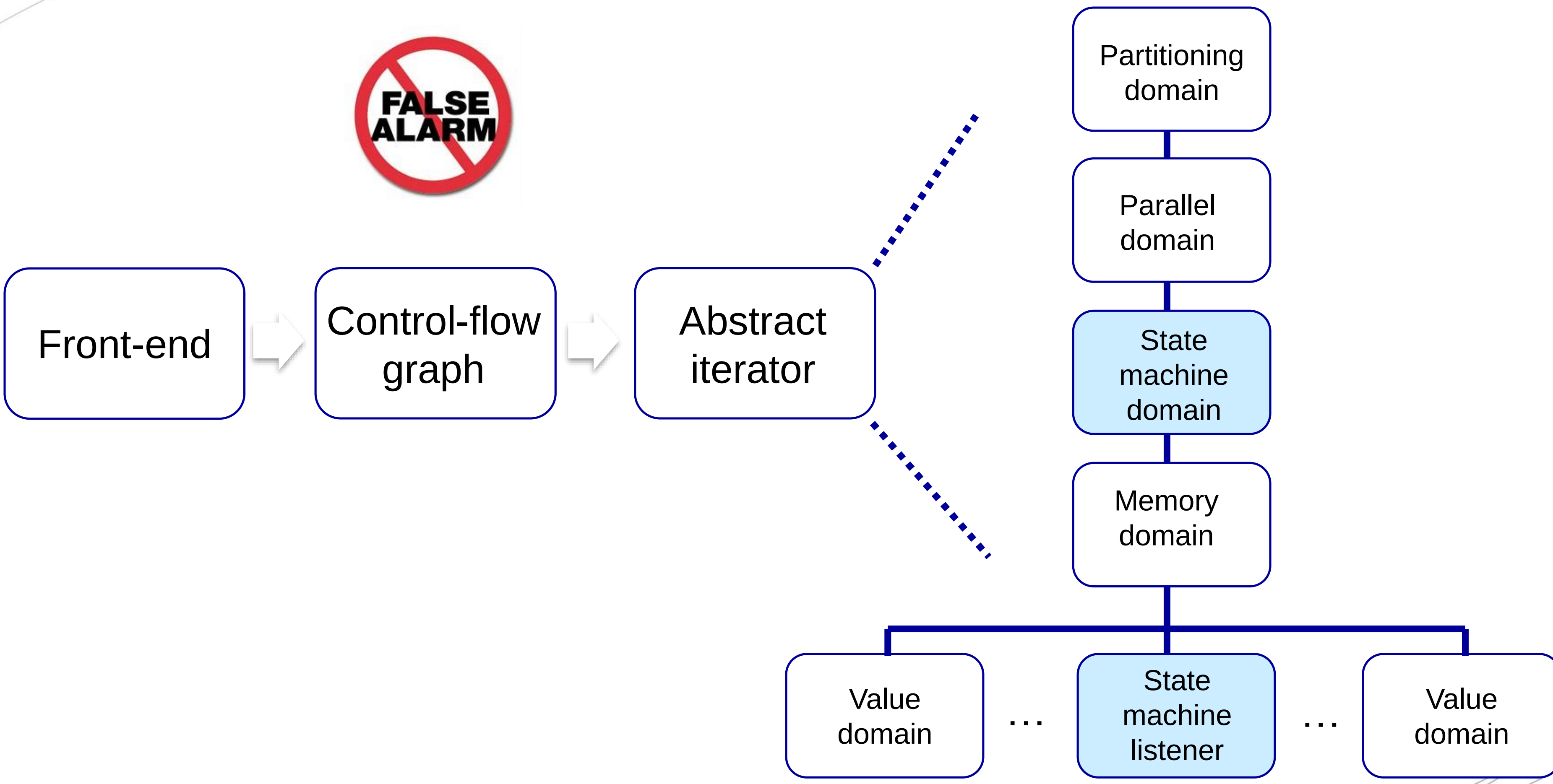
Runtime Errors and Data Races

- Abstract Interpretation-based static runtime error analysis
- Astrée detects all runtime errors* with few false alarms:
  - Array index out of bounds
  - Int/float division by 0
  - Invalid pointer dereferences
  - Uninitialized variables
  - Arithmetic overflows
  - Data races
  - Lock/unlock problems, deadlocks
  - Floating point overflows, Inf, NaN
  - Taint analysis (data safety / security), SPECTRE detection
  - \+ Floating-point rounding errors taken into account
  - \+ User-defined assertions, unreachable code, non-terminating loops
  - \+ Check coding guidelines (MISRA C/C++, CERT, CWE, ISO TS 17961)



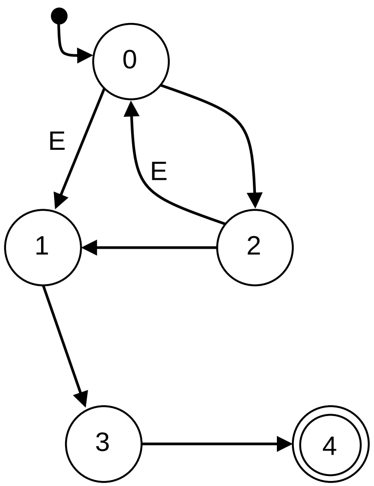
* Defects due to undefined / unspecified behaviors of the programming language

Design of Astrée



Finite State Machines: Example

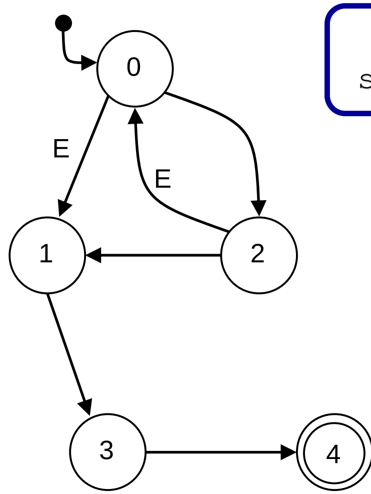
```
int *p; int state = 0;
while (1) {env_get(&E);
  switch (state) {
  case 0:
    if (E) state = 1;
    else state = 2;
    break;
  case 1:
    state = 3;
    p = &state;
    break;
  case 2:
    if (E) state = 0;
    else state = 1;
    break;
  case 3:
    *p = 4;
    break;
  case 4:
    return;
  }
}
```



“Normal” Analysis

```
int *p; int state = 0;
while (1) {env_get(&E);
  switch (state) {
  case 0:
    if (E) state = 1;
    else state = 2;
    break;
  case 1:
    state = 3;
    p = &state;
    break;
  case 2:
    if (E) state = 0;
    else state = 1;
    break;
  case 3:
    *p = 4;
    break;
  case 4:
    return;
  }
}
```

Iter 1	Iter 2	Iter 3	Iter 4
p:INVALID state:{0}	p:INVALID state:[0,2]	p:{INVALID, &state} State:[0,3]	p:{INVALID, &state} State:[0,4]
p:INVALID state:{1,2}	p:INVALID state:{1,2}	p:{INVALID, &state} state:{1,2}	p:{INVALID, &state} state:{1,2}
	p:&state state:{3}	p:&state state:{3}	p:&state state:{3}
	p:INVALID state:{0,1}	p:{INVALID, &state} state:{0,1}	p:{INVALID, &state} state:{0,1}
		p:{INVALID, &state} state:{3,4}	p:{INVALID, &state} state:{3,4}
			p:{INVALID, &state} state:{4}

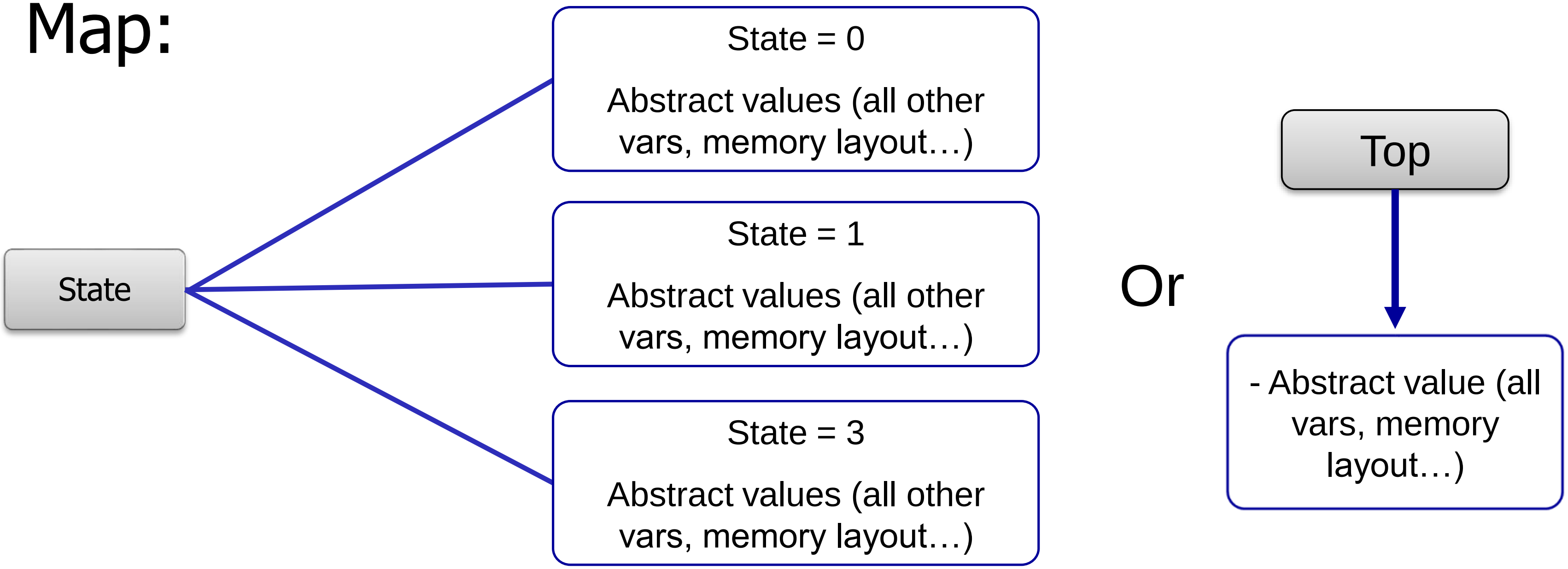


ALARM: Invalid pointer dereference

AbsInt

State Machine Domain

- Implements basic disjunction over states
- Map:



Transfer functions: applied to each leaf

▷ How do we cover all states and keep them disjoint?

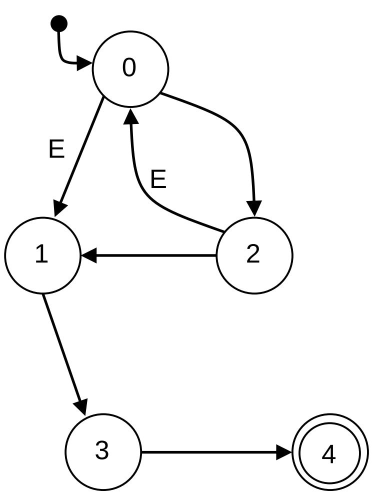
State Machine Listener Domain

- Dedicated domain, below memory layout domain
- Keeps track of memory blocks associated with state machine variable keys
  - Manual and/or automatic (heuristic) state variable detection
    - Start following variable (`__ASTREE_states_track`)
    - Stop following variable when merging all state machine states (`__ASTREE_states_merge`)
  - For each transfer function (assignment, memcpy,…), check if value changes for a state variable key
- Each time a state variable is modified
  - Compute new set of values
  - Re-compute disjunctions, join states with same values

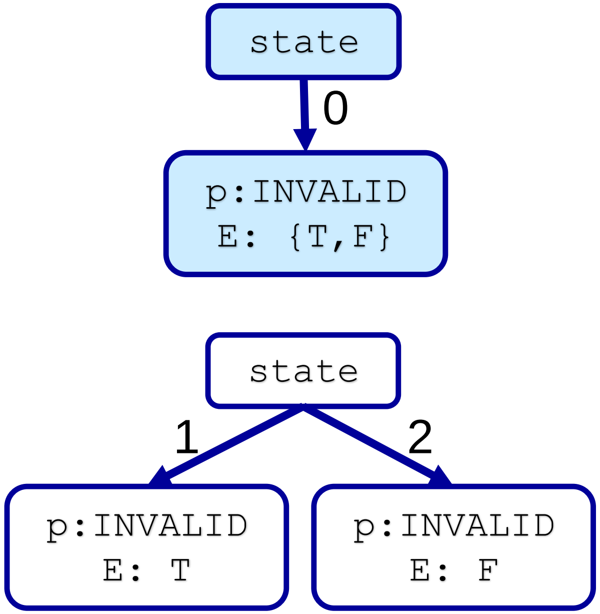
AbsInt

FSM Analysis

```
int *p; int state = 0;
while (1) {env_get(&E);
  switch (state) {
  case 0:
    if (E) state = 1;
    else state = 2;
    break;
  case 1:
    state = 3;
    p = &state;
    break;
  case 2:
    if (E) state = 0;
    else state = 1;
    break;
  case 3:
    *p = 4;
    break;
  case 4:
    return;
  }
}
```

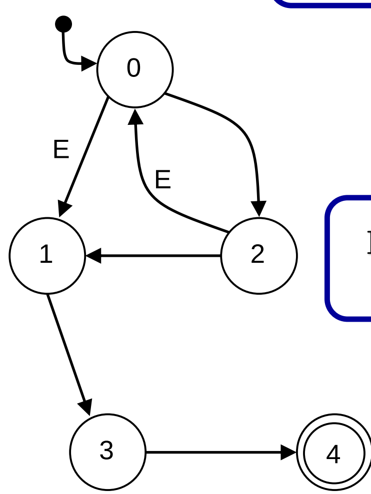


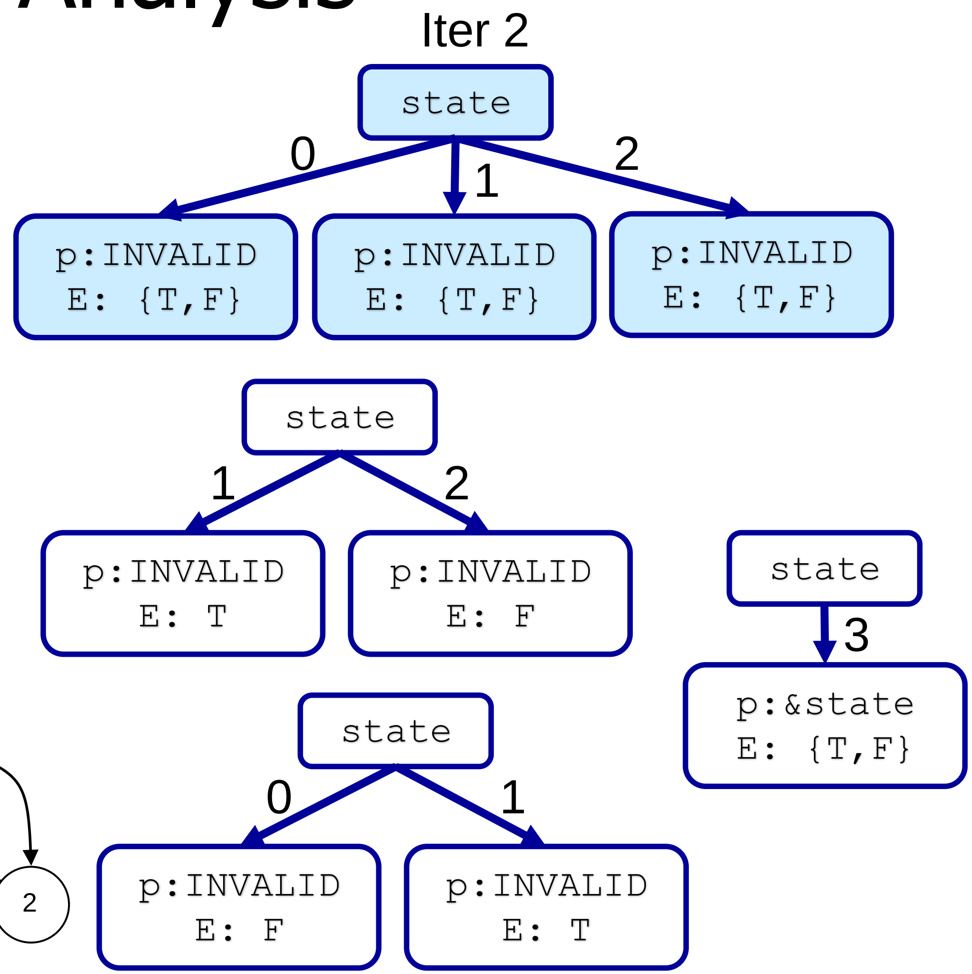
Iter 1



FSM Analysis

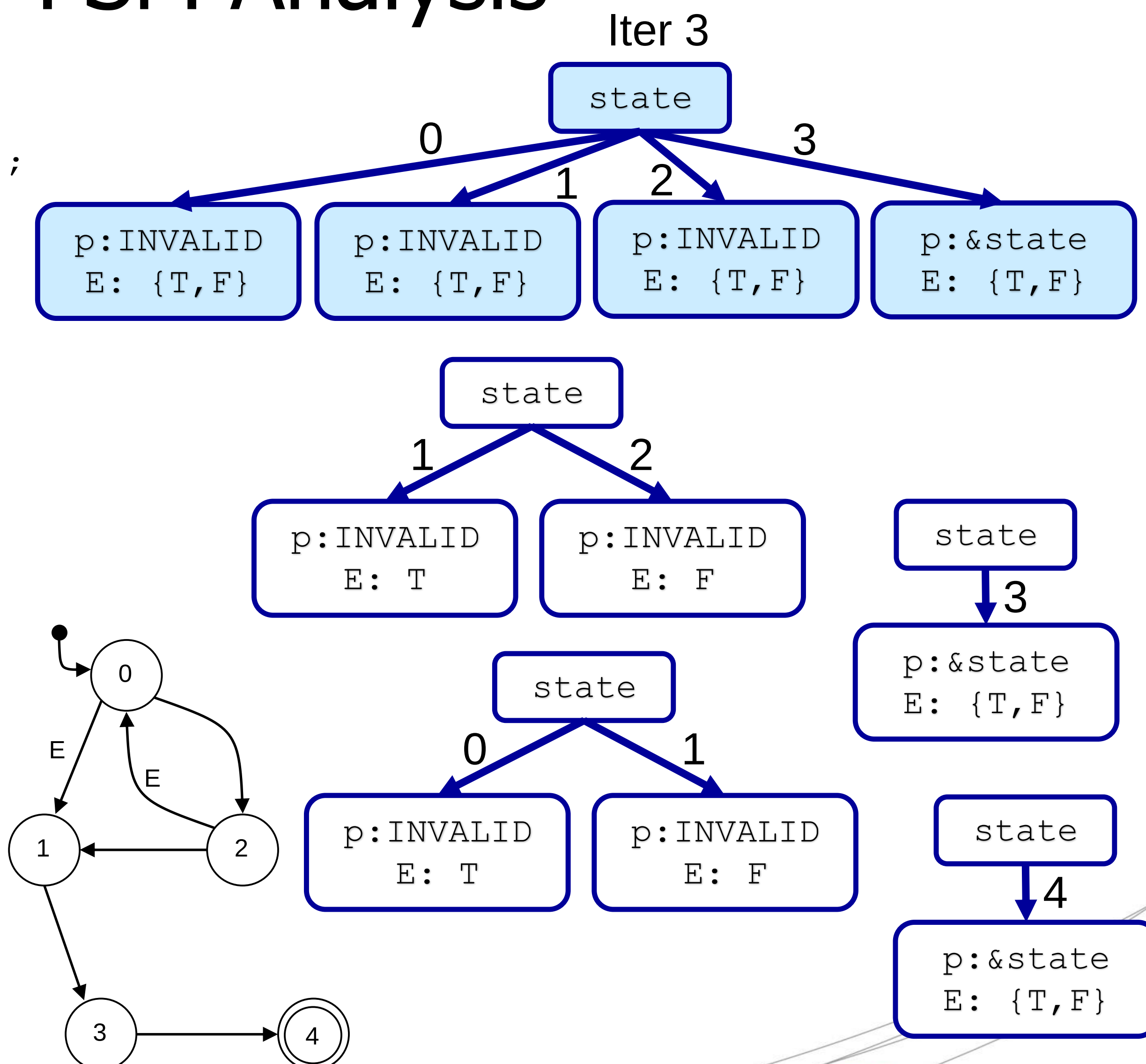
```
int *p; int state = 0;
while (1) {env_get(&E);
  switch (state) {
  case 0:
    if (E) state = 1;
    else state = 2;
    break;
  case 1:
    state = 3;
    p = &state;
    break;
  case 2:
    if (E) state = 0;
    else state = 1;
    break;
  case 3:
    *p = 4;
    break;
  case 4:
    return;
  }
}
```





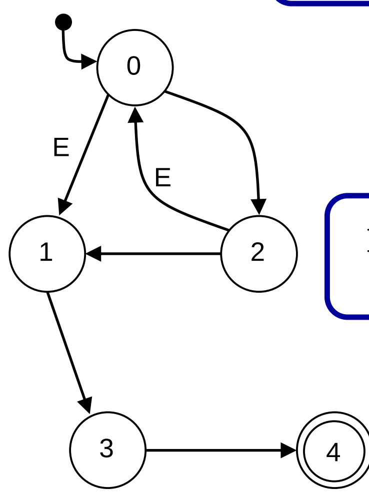
FSM Analysis

```
int *p; int state = 0;
while (1) {env_get(&E);
  switch (state) {
  case 0:
    if (E) state = 1;
    else state = 2;
    break;
  case 1:
    state = 3;
    p = &state;
    break;
  case 2:
    if (E) state = 0;
    else state = 1;
    break;
  case 3:
    *p = 4;
    break;
  case 4:
    return;
  }
}
```

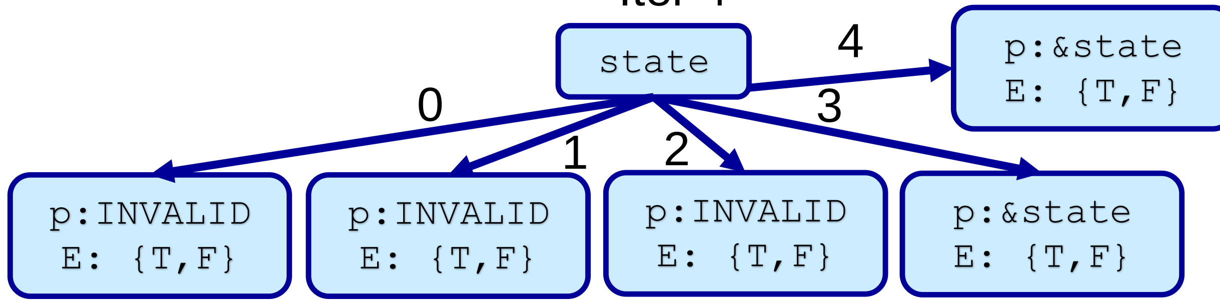


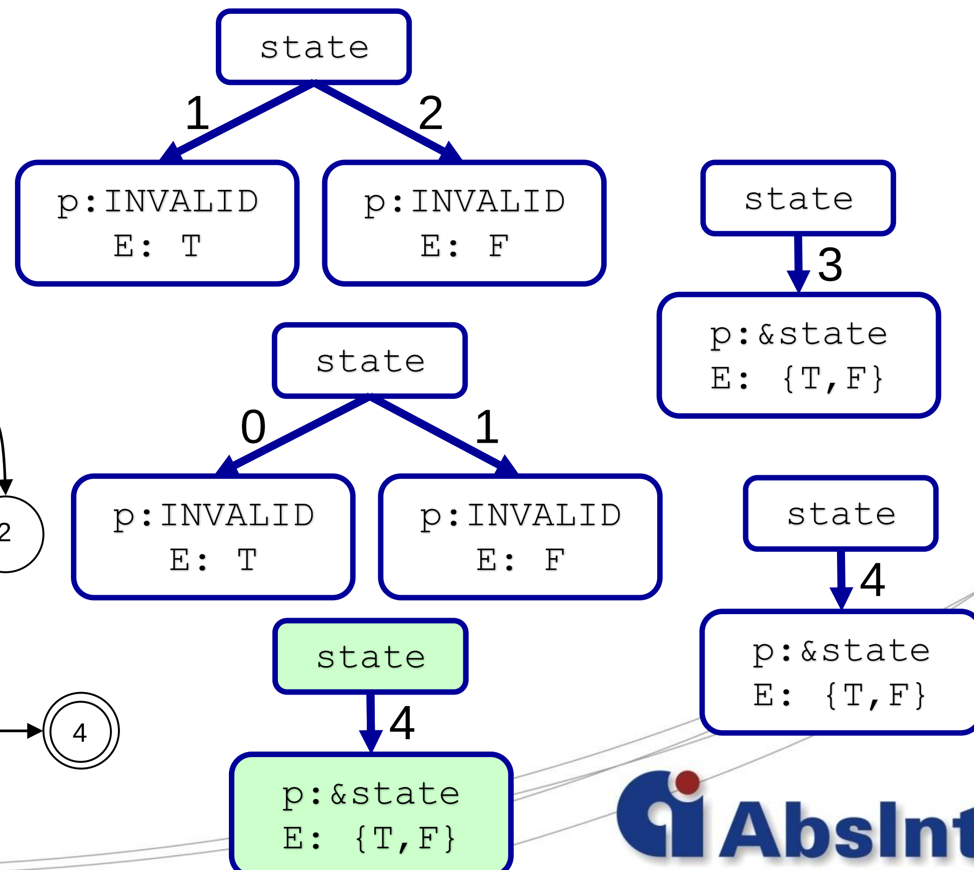
FSM Analysis

```
int *p; int state = 0;
while (1) {env_get(&E);
  switch (state) {
  case 0:
    if (E) state = 1;
    else state = 2;
    break;
  case 1:
    state = 3;
    p = &state;
    break;
  case 2:
    if (E) state = 0;
    else state = 1;
    break;
  case 3:
    *p = 4;
    break;
  case 4:
    return;
  }
}
```



Iter 4





Experimental Results

Benchmark	Code Size	#Errors		#Alarms		Memory		Time		#States
	(LOC)	wo/	w/	wo/	w/	wo/	w/	wo/	w/	max
B1 (I)	348 530	1	0	45	4	814	424	24'34"	9"	4
B2 (I)(*)	11 646	2	2	82	80	482	647	5'22"	8'50"	3
B3 (TL)	2 335	0	0	34	34	215	230	16"	3'15"	24
B4 (Sc)	4 442	0	0	15	3	156	159	2"	3"	3
B5 (I)(Sc)	8 733	0	0	57	48	173	243	6"	30"	14
B6 (I)	2 044 805	6	6	1787	1787	12 729	15 167	4h07'	3h32'	4

*: state machine automatically detected by Astrée

I: industrial code

TL: code generated by dSPACE TargetLink

Sc: code generated by SCADE

wo/: without FSM domain; w/: with FSM domain

- With FSM domain, zero false alarms due to imprecision caused by state machine code structures.
- Max observed increase in RAM: 40% (B5), max decrease: 48% (B1)
- Analysis time typically increases, but can also decrease as higher precision prevents spurious paths/values from being analyzed.

AbsInt

Taint Analysis

- Purpose: Static analysis to track flow of tainted values through program.
- Concepts:
  - Tainted source: origin of tainted values
  - Restricted sink: operands and arguments to be protected from tainted values
  - Sanitization: remove taint from value, e.g. by replacement or termination
- User interaction to identify tainted sources and sinks.
- Applications:
  - Information Flow (Confidentiality / Information Leaks)
  - Propagation of Error Values (Data and Control Flow)
  - Data Safety

Spectre Classes

- Transient execution attacks: transfer microarchitectural state changes caused by the execution of transient instructions (i.e., whose result is never committed to architectural state) to an observable architectural state.
  - Meltdown: transient out-of-order instructions after CPU exception
  - Spectre: exploit branch misprediction events

- Spectre types
  - Spectre-PHT: Pattern History Table ▷ Spectre V1, V1.1, SplitSpectre
  - Spectre-BTB: Brant Target Buffer ▷ Spectre V2
  - Spectre-STL: Store-to-Load Forwarding ▷ Spectre V4
  - Spectre-RSB: Return Stack Buffer ▷ ret2spec, Spectre-RSB

Vulnerable Code and Fix

```
ErrCode vulnerable1 (unsigned idx )
{
  if (idx >= arr1.size) {
    return E_INVALID_PARAMETER;
  }
  unsigned u1 = arr1.data[idx];
  ...
  unsigned u2 = arr2.data[u1];
  ...
}
```

Untrusted data (attacker-controlled)

Can be executed with out-of-range values after mis-predicted branches

Value read from `arr1` is used to index `arr2`. The memory access modifies the cache.

Timing attack can identify cache cell with hit, which leaks `u1`, ie., the contents of `arr1`.

Fix

```
ErrCode vulnerable1 (unsigned idx)
{
  if (idx >= arr1.size) {
    return E_INVALID_PARAMETER;
  }
  unsigned fidx = FENCEIDX(idx,arr1.size);
  ...
  unsigned u1 = arr1.data[fidx];
  ...
  unsigned u2 = arr2.data[u1];
  ...
}
```

`FENCEIDX` maps `idx` into the feasible array range.

Taint Analysis for Spectre

- Two taints: `controlled` and `dangerous`
- Manual tainting of user-controlled values as `controlled`
  - E.g.: all parameters of relevant OS functions

- Automatic detection of comparison of `controlled` values with bounds

⇨ Taint automatically changed from `controlled` to `dangerous`

- Remove `dangerous` taint at end of speculative execution window. Architecture-independent solution:

⇨ Automatic reset to `controlled` at control flow join

Example

```
volatile int controlled;
__ASTREE_volatile_input((controlled; [1,2]));
int victim_function( size_t x ) {
  if ( x < array1_size ) {
    temp &= array2 [array1[ x ] * 512];
  }
  return x;
}
void main(){
  unsigned int val, retval;
  init(&val);          //reads val from the environment
  __ASTREE_taint((val; controlled));
  retval = victim_function( val );
}
```

ALARM: Spectre vulnerability

- No complete protection but attack surface can be reduced
- Almost no overhead to pure run-time error analysis

Conclusion

- In safety-critical systems the absence of safety and security hazards has to be demonstrated.
- Sound static analysis crucial for safety and security
  - Absence of critical code defects can be proven
  - No runtime errors: "pretty good security‟
  - Sound data and control coupling
- Low false alarm rate and low analysis time crucial
  - Sophisticated abstract domains to achieve zero-false-alarm goal
  - Example: novel FSM domain for fast and precise analysis of finite state machines
- Taint analysis based on sound analysis framework
  - User-configurable impact analysis (data corruption)
  - Spectre detection

email: info@absint.com
http://www.absint.com