SAT Solver architectures
Presentation at FMV, Linz

Mate Soos
Security Research Labs

9th of February 2011
Motivations and goals

Motivations

- Architectural choices are not simple
- Rarely talked about, but matter a lot

Goals

- Present the different choices
- Shed light on their interactions
Table of Contents

Context

Architectures

Conclusions
Outline

Context

SAT Solvers
Challenges

Architectures

Main loop
Simplification algos.
Multi-threaded/distributed solving
Extra functionalities

Conclusions
SAT solvers

Input:
- CNF, an “and of or-s”:
  \((x_1 \vee \neg x_3) \land (\neg x_2 \lor x_3) \land (x_1 \lor x_2)\)
- Crypto-problem needs conversion
- E.g. \(a \oplus bc \oplus d = 0\) needs internal var for \(bc\)

Uses DPLL(\(\varphi\)) algorithm

1. If (formula \(\varphi\) trivial) return SAT/UNSAT
2. \(\text{ret} \leftarrow \text{DPLL}(\varphi \text{ with } v \leftarrow \text{true})\)
3. If (ret = SAT) return SAT
4. \(\text{ret} \leftarrow \text{DPLL}(\varphi \text{ with } v \leftarrow \text{false})\)
5. If (ret = SAT) return SAT
6. return UNSAT
Example search tree
Main challenges

Low-level constraints
- Efficient use of memory arch.: cache, page faults, etc.
- Must work well in multi-threaded: locks, semaphores, etc.
- Compiled code is of importance: compilation issues

High-level problems
- Highly theoretical approaches
- Complex interactions between algorithms
- Randomised algorithms
- No clear visual or statistical indicators
Additional challenges

Cut-offs

- Some algos. can take *far* too long to finish
- Limit time... but solver must remain deterministic
- Make the best of time available

Validity (bugs)

- UNSAT is rarely proven — solution hard to check
- Internal state very large — infection hard to trace
- Problems very heterogeneous – hidden bugs
Outline

Context
- SAT Solvers
- Challenges

Architectures
- Main loop
- Simplification algos.
- Multi-threaded/distributed solving
- Extra functionalities

Conclusions
Elements to add

Main elements
- DPLL
- Branch literals
- Restarts
- Learnt clause cleaning
- Thread synchronisation

Simplification algo
- Equivalent literals
- Clause subsumption
- Variable elimination
- ...
Must be efficient (70% of time)
  - Bit stuffing, hand-rolled memory managers, etc.
  - Prefetching, lazy algos
  - Special treatment of small clauses

Completeness vs. speed
  - Re-compute glues at propagation?
  - Extended impl. graph?
Learnt clauses & restarts

Options
- Geom./Luby rest. + clause activities: for packed problems
- Glue-based dyn rest. + glues: for large industrial problems
- Agility-based dyn rest. + glues: for both, but not perfect
- Outer-inner variations of above

Problems
- Important choice: cache behav., complexity of impl.
- Support multiple vs. speed
- Which to choose if multiple supported
Scheduling of simplification algos.

In what order?

- They interact...
- Cannot try all combinations: \( n! \)
- Intuition/habits/experience

When?

- Once at start-up: e.g. XOR extraction
- Once every X conflicts: e.g. failed lit. probing
- If needed: SCC if new binary clauses
- AT every conflict: confl. minimisation
Organising simplification algos.

Generally
- Some need occurrence lists: detach & reattach
- Some need to propagate w/o conflict analysis
- Some benefit from each other

In CryptoMiniSat
- Group 1: detached long clauses, occurrence lists
- Group 2: Simplified propagation, re-use of propagated values
Multi-threaded / Distributed solving

Do something different (heterogeneity)
- Support multiple restarts/activities
- Range of magic constants $\rightarrow$ more testing

Do the same, faster (sharing)
- Forced var setting — redundancy check: $O(1)$
- Binary clauses — redundancy check: $P$
- Tertiary clauses — redundancy check: $\text{co-NP}$?
- Larger clauses — provably needed
- What else to share?
Extra functionalities

Use solver as a

- Preprocessor
- Library for abstraction-refinement
- Multi-threaded and distributed variation of above
- Static analyser
- Dynamic analyser
- Accumulator of knowledge
Outline

Context
- SAT Solvers
- Challenges

Architectures
- Main loop
- Simplification algos.
- Multi-threaded/distributed solving
- Extra functionalities

Conclusions
Conclusions

- SAT solvers are complex
- Takes lots of time to write one
- Practise makes perfect
- But practise leads to habits
- Difficult to re-invent everything all the time