Using SAT Solvers for Cryptographic Problems

Presentation at Microsoft Research Centre, Cambridge

Mate Soos

UPMC LIP6, PLANETE team INRIA, SALSA Team INRIA

5th of November 2010

Motivations and goals

Motivations

- Not clear: When SAT solvers effective in crypto?
- Grobner basis?
- Brute-force?

Goals:

- Show differences between algorithms
- Demonstrate practical use-cases

Table of Contents

Outline

Cryptography

Types:

- Symmetric: key + plaintext $\rightarrow$ ciphertext
- Hash functions: text fingerprinting
- Asymmetric: signature, private encoding

Complexity:

- Theory: Brute force best attack, rarely proven
- Clean-room attacks: statistical, complexity-based
- Side-channel attacks: passive (EM radiation) /active (fuzzing)

Grobner basis: Faugere's F4/F5

- Input set of polys $a \oplus bc \oplus d = 0$
- Grobner basis by echelonisation of large matrix
- Incrementally, as matrix is *large*
- F5: no redundant calculation

SAT solvers

Input:

- CNF, an "and of or-s": $(x_1 \vee \neg x_3) \quad \wedge \quad (\neg x_2 \vee x_3) \quad \wedge \quad (x_1 \vee x_2)$
- Crypto-problem needs conversion
- E.g. $a \oplus bc \oplus d = 0$ needs internal var for bc

Uses DPLL(φ) algorithm

1. If (formula φ trivial) return SAT/UNSAT
2. ret $\leftarrow$ DPLL(φ with $v \leftarrow$ `true`)
3. If (ret $=$ SAT) return SAT
4. ret $\leftarrow$ DPLL(φ with $v \leftarrow$ `false`)
5. If (ret $=$ SAT) return SAT
6. return UNSAT

Brute force

Theory:

- Input is set of operations on key, plaintext
- Execute set of operations 2^k times
- On average: 2^{k-1} tries

Practise:

- Some keys may be eliminated (e.g. DES)
- Uses CUDA, FPGA
- Execution *very* optimised

Outline

Grobner basis vs. SAT solvers

- Upper bound of both: doubly exponential
- Practical behaviour of both: *much* better than bound

- Grobner basis: lower bound can sometimes be proven
- But practise is still much faster than theory
- No such lower bound for SAT: harder to argument

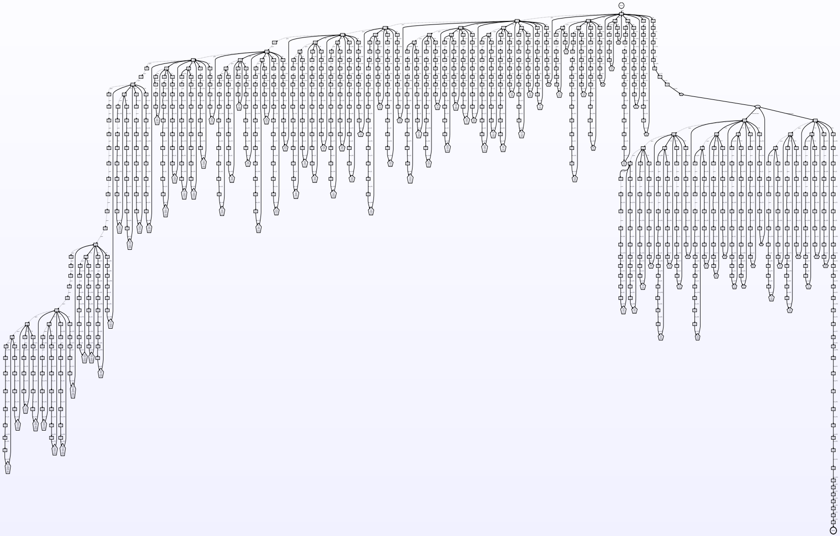
SAT solvers vs. Brute force

- Both go through a search tree

- Brute force avoids same parts through division
- SAT avoids same parts through learnt clauses

- Brute force re-computes everything every time
- SAT solvers backjump, keeping partial state
- Internal variables are used to keep state

Example search tree



SAT solvers vs. Brute force

- For crypto-problems, finding UNSAT = 2*SAT time
- Just like Brute force

- Interesting, because highlights search-tree approach
- Hard to argument from resolution-tree approach

Outline

Medium/low complexity systems

When they arise:

- Unexpectedly easy or low budget: HFE, HiTag, Mifare
- Side-channel: added information makes system easy

Solving them:

- Brute force: if key small (HiTag2)
- Grobner basis: for hidden low-complexity (HFE)
- SAT: for information-rich problems (side-channel info)

Why not Grobner basis?

- Uses PC with tens of GB of memory
- Algorithm start-up is non-trivial (minutes/hours)
- Details of algorithm unknown: harder to publish
- Proprietary: Magma expensive

Why SAT solvers?

Learnt clauses:

- Act as memory
- Apply to different parts of the search tree

Lazy data structures:

- Fast partial back-tracking
- Keep partially computed values in memory

Variable activity heuristics:

- Find good points of entry
- E.g. key bits, shift register states, etc.

Outline

1. Context
   - Cryptography
   - Solving methods
2. Comparison of solving methods
   - F5 vs. SAT
   - SAT vs. Brute force
3. Practical problem solving
   - An example problem
   - Why SAT Solvers?
4. Conclusions

Concluding remarks:

- SAT: low-complexity ciphers, side-channel attacks
- Grobner basis: HFE, multivariate crypto schemes

Future work:

- Integrate the two
- As pre-, or post-processors to each other
- As in-processors (e.g. Gauss-elim. in CryptoMS)

Thank you for your time

Any questions?