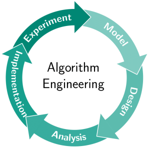
Karlsruher Institut für Technologie



Trusted Scalable SAT Solving with on-the-fly LRAT Checking

SAT 2024, Pune, India

Dominik Schreiber | August 22, 2024

Motivation

Distributed clause-sharing solvers push the frontier of feasible problems.

- Many sequential CDCL solvers run in parallel
- Careful exchange of useful conflict clauses
- Mean speedup of 419 @ 3072 cores for difficult instances [SS24]

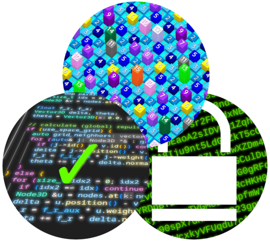
Motivation

Distributed clause-sharing solvers push the frontier of feasible problems.

- Many sequential CDCL solvers run in parallel
- Careful exchange of useful conflict clauses
- Mean speedup of 419 @ 3072 cores for difficult instances [SS24]

Proofs of unsatisfiability are central for trust in SAT solving.

- Model checking critical software? UNSAT claims safety!
- Suffices to trust independent proof checker (+ underlying technology)



Motivation

Karlsruhe Institute of Technology

Distributed clause-sharing solvers push the frontier of feasible problems.

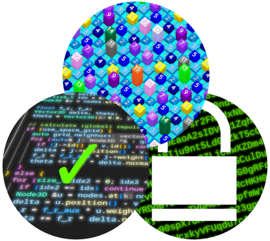
- Many sequential CDCL solvers run in parallel
- Careful exchange of useful conflict clauses
- Mean speedup of 419 @ 3072 cores for difficult instances [SS24]

Proofs of unsatisfiability are central for trust in SAT solving.

- Model checking critical software? UNSAT claims safety!
- Suffices to trust independent proof checker (+ underlying technology)

Parallel & distributed solvers are harder to trust than sequential solvers.

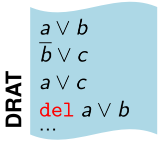
- Large technology stack leaves more room for bugs, errors
- More difficult and expensive to test rigorously
- Fragile – a single bit flip in a clause can induce a wrong result



The Story Thus Far

Producing proofs from parallel clause sharing is challenging.

- Popular DRAT format does not scale in parallel settings [HMP14; FB22]



The Story Thus Far

Producing proofs from parallel clause sharing is challenging.

- Popular DRAT format does not scale in parallel settings [HMP14; FB22]
- Explicit dependency data in LRAT format allows for feasible distributed proof production [Mic+23]

DRAT

$a \lor b$
$\overline{b} \lor c$
$a \lor c$
del $a \lor b$
…

LRAT

$1765 : a \lor b \mid 823, 1277$
$1766 : \overline{b} \lor c \mid 1338, 54$
$1767 : a \lor c \mid 1765, 1766$
del 1765
…

The Story Thus Far

Producing proofs from parallel clause sharing is challenging.

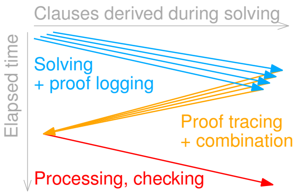
- Popular DRAT format does not scale in parallel settings [HMP14; FB22]
- Explicit dependency data in LRAT format allows for feasible distributed proof production [Mic+23]
  1. Write individual partial proofs during solving
  2. Rewind solving + sharing, funnel required derivations into single file
  3. Check combined proof file

DRAT

$a \lor b$
$\overline{b} \lor c$
$a \lor c$
del $a \lor b$
…

LRAT

$1765 : a \lor b \mid 823, 1277$
$1766 : \overline{b} \lor c \mid 1338, 54$
$1767 : a \lor c \mid 1765, 1766$
del 1765
…



The Story Thus Far

Karlsruhe Institute of Technology

Producing proofs from parallel clause sharing is challenging.

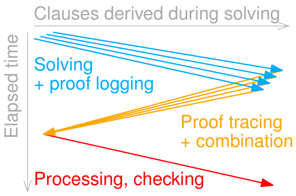
- Popular DRAT format does not scale in parallel settings [HMP14; FB22]
- Explicit dependency data in LRAT format allows for feasible distributed proof production [Mic+23]
  1. Write individual partial proofs during solving
  2. Rewind solving + sharing, funnel required derivations into single file
  3. Check combined proof file
- **Bottleneck:** sequential assembly and checking of monolithic proof
  - Throttled by I/O bandwidth at final process
  - Sometimes hundreds of Gigabytes of proof information
  - Proof production + checking @ 1520 cores takes $\approx 3\times$ solving time (latest setup – submitted to JAR)
  - Intuition "*If solving fits into RAM, checking will as well*" no longer holds

DRAT

$a \vee b$
$\overline{b} \vee c$
$a \vee c$
del $a \vee b$
…

LRAT

$1765 : a \vee b \mid 823, 1277$
$1766 : \overline{b} \vee c \mid 1338, 54$
$1767 : a \vee c \mid 1765, 1766$
del 1765
…

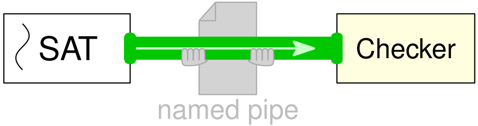


On-the-fly Checking with Sequential Solvers

Marijn Heule: Since LRAT checking is so efficient, we can feasibly do it in realtime!

```
mkfifo lratproof.pipe  // create “pipe” file

// Solve & check concurrently via pipe
./solver input.cnf lratproof.pipe &
   ./lrat-check input.cnf lratproof.pipe
```



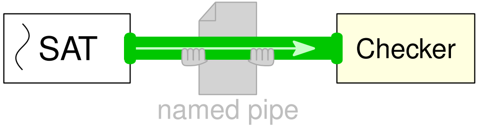
- No disk I/O, direct inter-process communication
- Program code indistinguishable from plain file I/O (only difference: `mkfifo`)

On-the-fly Checking with Sequential Solvers

Marijn Heule: Since LRAT checking is so efficient, we can feasibly do it in realtime!

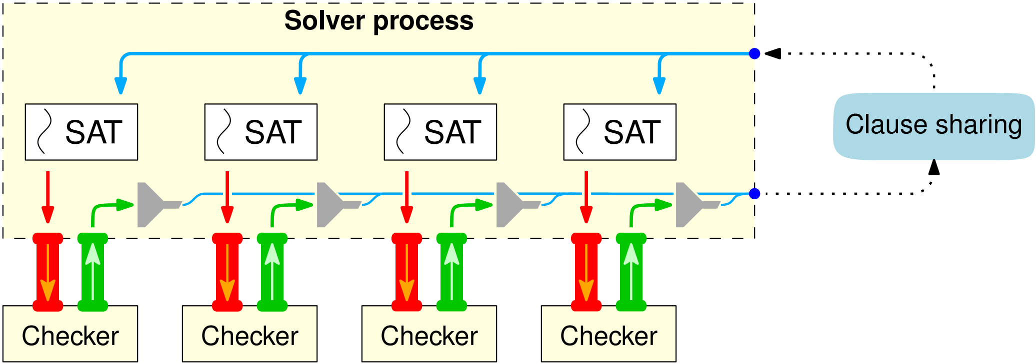
```
mkfifo lratproof.pipe  // create “pipe” file

// Solve & check concurrently via pipe
./solver input.cnf lratproof.pipe &
   ./lrat-check input.cnf lratproof.pipe
```

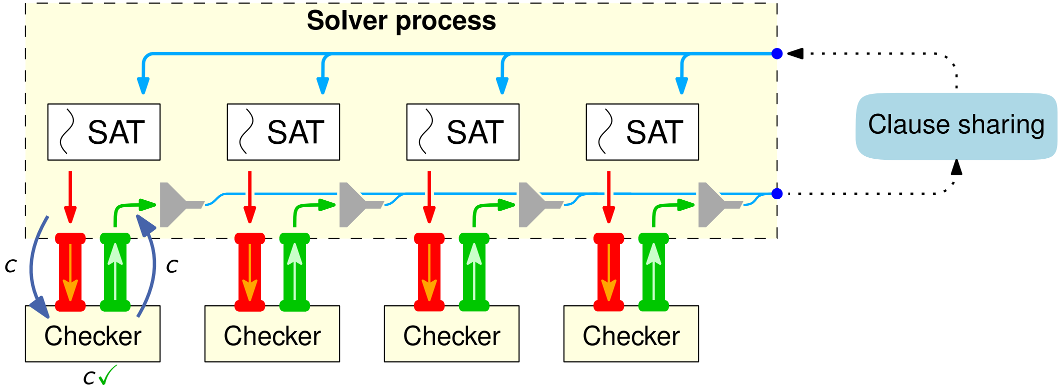


- No disk I/O, direct inter-process communication
- Program code indistinguishable from plain file I/O (only difference: `mkfifo`)
- Does not yield a persistent artifact to validate by independent parties

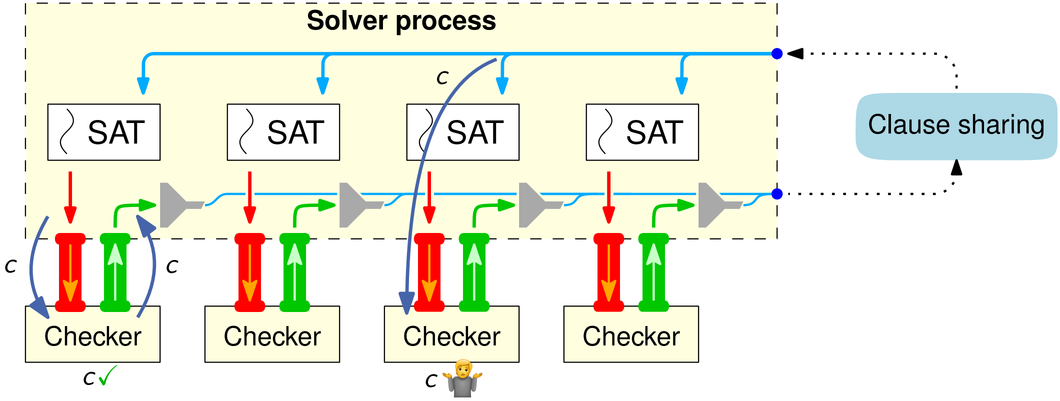
A First Parallel & Distributed Setup



A First Parallel & Distributed Setup



A First Parallel & Distributed Setup



A Question of Trust

Which components do we still need to trust?

- Parser (reads correct formula correctly)

A Question of Trust

Karlsruhe Institute of Technology

Which components do we still need to trust?

- Parser (reads correct formula correctly)
- Checker process (performs sound LRAT checking and responds accordingly)

A Question of Trust

Which components do we still need to trust?

- Parser (reads correct formula correctly)
- Checker process (performs sound LRAT checking and responds accordingly)
- Solver process (does not forward unchecked clauses to sharing)

A Question of Trust

Which components do we still need to trust?

- Parser (reads correct formula correctly)
- Checker process (performs sound LRAT checking and responds accordingly)
- Solver process (does not forward unchecked clauses to sharing)
- Distributed communication (does not compromise / corrupt / truncate a message)

A Question of Trust

Which components do we still need to trust?

- Parser (reads correct formula correctly)
- Checker process (performs sound LRAT checking and responds accordingly)
- Solver process (does not forward unchecked clauses to sharing)
- Distributed communication (does not compromise / corrupt / truncate a message)
- The seq. SAT solver (doesn't forward an unsound clause as an axiom to the checker)

A Question of Trust

Karlsruhe Institute of Technology

Which components do we still need to trust?

- Parser (reads correct formula correctly) — I/O error?
- Checker process (performs sound LRAT checking and responds accordingly)
- Solver process (does not forward unchecked clauses to sharing) — Application bug?
- Distributed communication (does not compromise / corrupt / truncate a message) — MPI bugs?
- The seq. SAT solver (doesn't forward an unsound clause as an axiom to the checker)

A Question of Trust

Karlsruhe Institute of Technology

Which components do we still need to trust?

- Parser (reads correct formula correctly) — I/O error?
- Checker process (performs sound LRAT checking and responds accordingly)
- Solver process (does not forward unchecked clauses to sharing) — Application bug?
- Distributed communication (does not compromise / corrupt / truncate a message) — MPI bugs?
- The seq. SAT solver (doesn't forward an unsound clause as an axiom to the checker)

In terms of limiting our "trusted parties", we haven't really gained anything.

A Question of Trust

Karlsruhe Institute of Technology

Which components do we still need to trust?

- Parser (reads correct formula correctly) — I/O error?
- Checker process (performs sound LRAT checking and responds accordingly)
- Solver process (does not forward unchecked clauses to sharing) — Application bug?
- Distributed communication (does not compromise / corrupt / truncate a message) — MPI bugs?
- The seq. SAT solver (doesn't forward an unsound clause as an axiom to the checker)

In terms of limiting our "trusted parties", we haven't really gained anything.

Goal: Only need to trust the parser and checkers, nothing else!

Signatures (1/2)

Clause $\xrightarrow{\mathcal{S}}$ Signature

ID: 159514 | Lits: $4 \lor \overline{163} \lor \overline{145} \lor \overline{28} \lor 158$ → `d12e6a68fc3456e95d64a735555783d6`

Assumption: Parser and checkers know a "secret" signature function $\mathcal{S}$

Signatures (1/2)

Clause

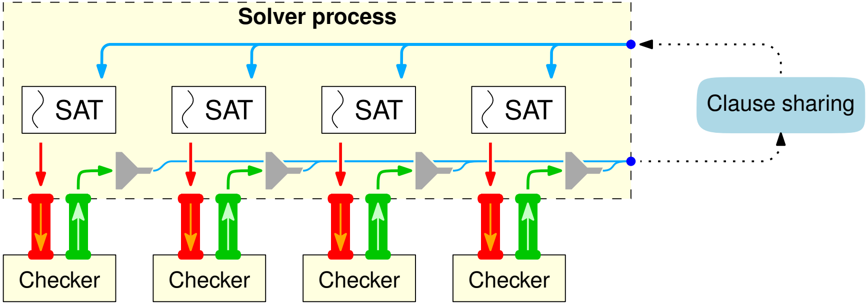
ID: 159514 | Lits: $4 \vee \overline{163} \vee \overline{145} \vee \overline{28} \vee 158$

$\mathcal{S}$ →

Signature

`d12e6a68fc3456e95d64a735555783d6`

Assumption: Parser and checkers know a "secret" signature function $\mathcal{S}$



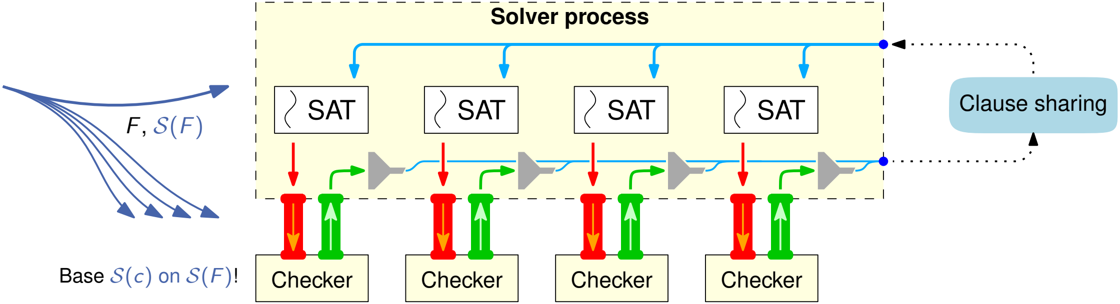
Signatures (1/2)

Clause $\xrightarrow{\mathcal{S}}$ Signature

ID: 159514 | Lits: $4 \lor \overline{163} \lor \overline{145} \lor \overline{28} \lor 158$

`d12e6a68fc3456e95d64a735555783d6`

Assumption: Parser and checkers know a "secret" signature function $\mathcal{S}$



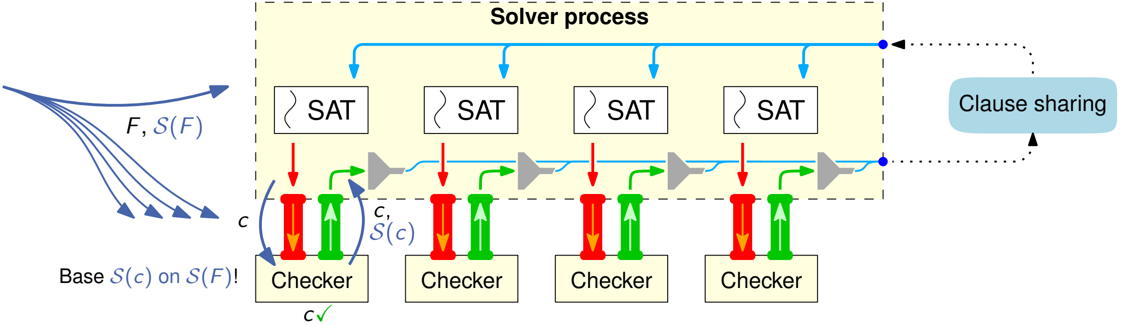
Signatures (1/2)

Clause → $\mathcal{S}$ → Signature

ID: 159514 | Lits: $4 \vee \overline{163} \vee \overline{145} \vee \overline{28} \vee 158$

`d12e6a68fc3456e95d64a735555783d6`

Assumption: Parser and checkers know a “secret” signature function $\mathcal{S}$



Signatures (1/2)

Clause

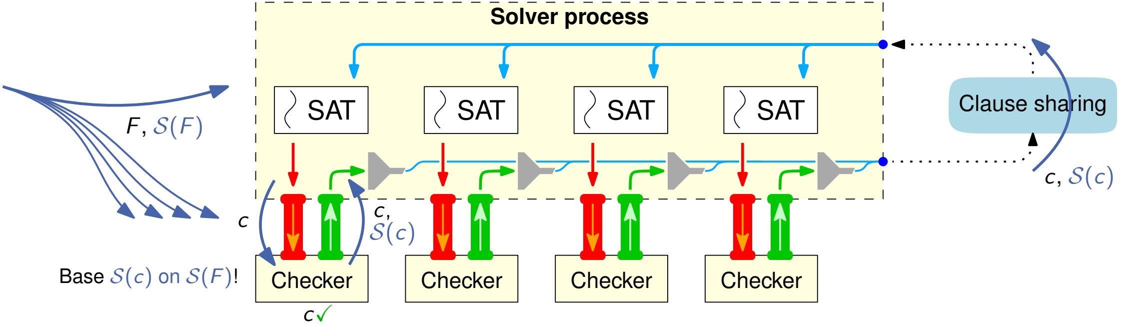
ID: 159514 | Lits: $4 \lor \overline{163} \lor \overline{145} \lor \overline{28} \lor 158$

$\mathcal{S}$

Signature

`d12e6a68fc3456e95d64a735555783d6`

Assumption: Parser and checkers know a “secret” signature function $\mathcal{S}$

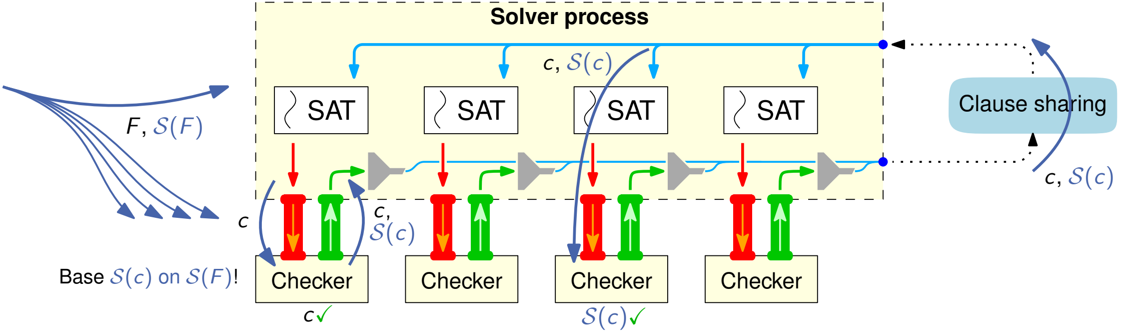


Signatures (1/2)

Clause	Signature
ID: 159514 | Lits: $4 \lor \overline{163} \lor \overline{145} \lor \overline{28} \lor 158$	`d12e6a68fc3456e95d64a735555783d6`

$\xrightarrow{\mathcal{S}}$

Assumption: Parser and checkers know a “secret” signature function $\mathcal{S}$



Signatures (2/2)

- General framework: **Message Authentication Code** (MAC)
  - Allows trusted parties to sign and validate messages using shared secret K
  - Ensures authenticity (no confidentiality)

Signatures (2/2)

- General framework: **Message Authentication Code** (MAC)
  - Allows trusted parties to sign and validate messages using shared secret K
  - Ensures authenticity (no confidentiality)
- Chosen function: **SipHash** [AB12] – keyed hash function $\mathcal{S}(x) := H_K(x)$
  - Fast – only uses add-rotate-xor (ARX)
  - Popular, battle-tested, scrutinized

Signatures (2/2)

- General framework: **Message Authentication Code** (MAC)
  - Allows trusted parties to sign and validate messages using shared secret K
  - Ensures authenticity (no confidentiality)
- Chosen function: **SipHash** [AB12] – keyed hash function $\mathcal{S}(x) := H_K(x)$
  - Fast – only uses add-rotate-xor (ARX)
  - Popular, battle-tested, scrutinized
- Only trusted processes (parser, checkers) may know K
  - Ensure K is present only in memory space of trusted processes
  - Current setup: K is hard-compiled into trusted processes

Signatures (2/2)

- General framework: **Message Authentication Code** (MAC)
  - Allows trusted parties to sign and validate messages using shared secret K
  - Ensures authenticity (no confidentiality)
- Chosen function: **SipHash** [AB12] – keyed hash function $\mathcal{S}(x) := H_K(x)$
  - Fast – only uses add-rotate-xor (ARX)
  - Popular, battle-tested, scrutinized
- Only trusted processes (parser, checkers) may know K
  - Ensure K is present only in memory space of trusted processes
  - Current setup: K is hard-compiled into trusted processes

$$\mathcal{S}(F) := H_K(F)\,, \quad \mathcal{S}(c) := H_K\big(id(c) \,||\, c \,||\, \mathcal{S}(F)\big)\,, \quad \mathcal{S}(\bot) := H_K\big(20 \,||\, \mathcal{S}(F)\big)$$

Signatures (2/2)

- General framework: **Message Authentication Code** (MAC)
  - Allows trusted parties to sign and validate messages using shared secret K
  - Ensures authenticity (no confidentiality)
- Chosen function: **SipHash** [AB12] – keyed hash function $\mathcal{S}(x) := H_K(x)$
  - Fast – only uses add-rotate-xor (ARX)
  - Popular, battle-tested, scrutinized
- Only trusted processes (parser, checkers) may know K
  - Ensure K is present only in memory space of trusted processes
  - Current setup: K is hard-compiled into trusted processes

$$\mathcal{S}(F) := H_K(F \mid\mid 0_{(2\text{ bytes})}) , \quad \mathcal{S}(c) := H_K\big(id(c) \mid\mid c \mid\mid \mathcal{S}(F)\big) , \quad \mathcal{S}(\bot) := H_K\big(20_{(1\text{ byte})} \mid\mid \mathcal{S}(F)\big)$$

Confidence

What breaks our approach?

Obtain $\mathcal{S}(\bot)$ for satisfiable F

Confidence

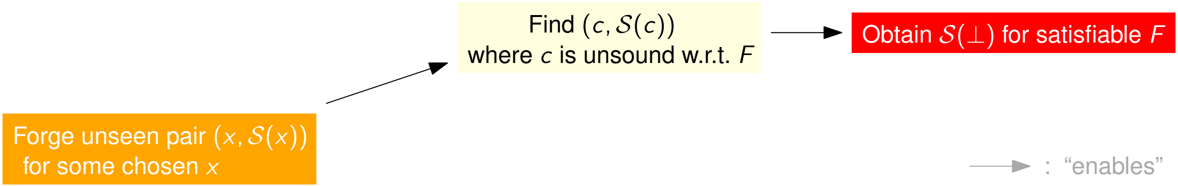
What breaks our approach?

Find $(c, \mathcal{S}(c))$ where c is unsound w.r.t. F → Obtain $\mathcal{S}(\perp)$ for satisfiable F

→ : "enables"

Confidence

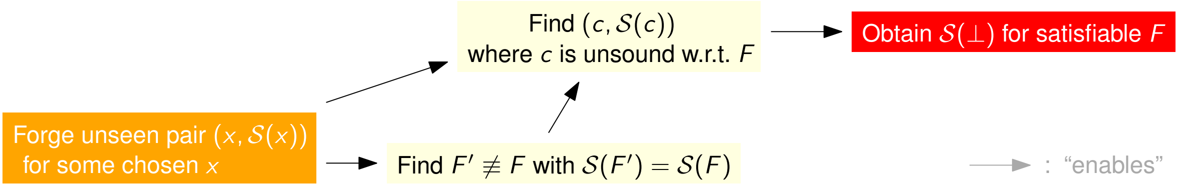
What breaks our approach?



Confidence

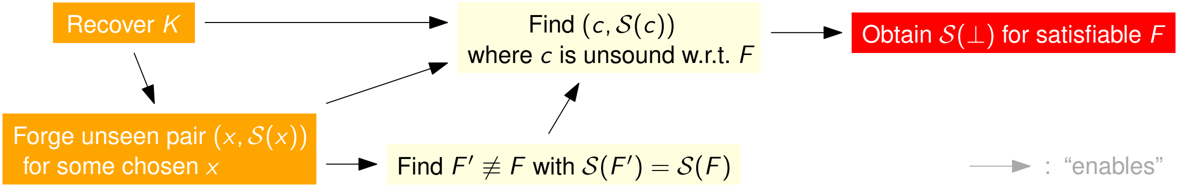
Karlsruhe Institute of Technology

What breaks our approach?



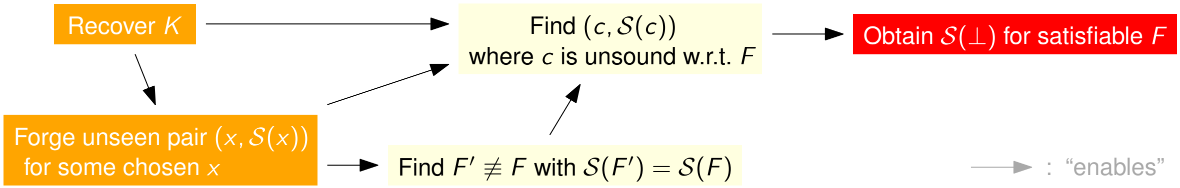
Confidence

What breaks our approach?



Confidence

What breaks our approach?

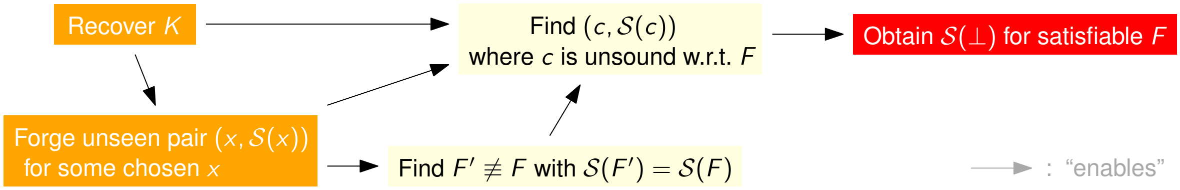


Security Claims of 128-bit SipHash

Forging a previously unseen pair $(x, \mathcal{S}_K(x))$ succeeds with probability $2^{-128} \approx 10^{-38}$.
Recovering K succeeds with probability 2^{-128}.

Confidence

What breaks our approach?



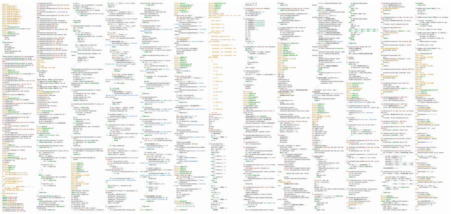
Security Claims of 128-bit SipHash

Forging a previously unseen pair $(x, \mathcal{S}_K(x))$ succeeds with probability $2^{-128} \approx 10^{-38}$.
Recovering K succeeds with probability 2^{-128}.

Intuition: Inadvertent bugs / errors / faults during solving "can't do better" than deliberate attacks!

Implementation

Implementation



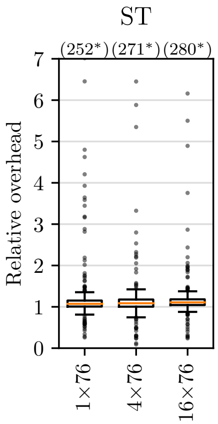
- Distributed framework: MALLOBSAT [SS24]
- Sequential solver: CADICAL with LRAT output [PFB23]
- Trusted modules: Parser, checker, confirmer
  - Confirmer takes F and $\mathcal{S}(\bot)$, validates $\mathcal{S}(\bot)$
  - Overall $\approx$ 1k effective lines of C99 code

Setup

- $\leq$ 32 compute nodes of HPC cluster HoreKa
  - Per node: 2×38 cores (76 hardware threads), 256 GB RAM
- SAT Competition 2023 benchmarks
- Time limits: 300 s wallclock time for solving,
  1500 s for postprocessing + checking

Overhead Relative to Proof-free Solving

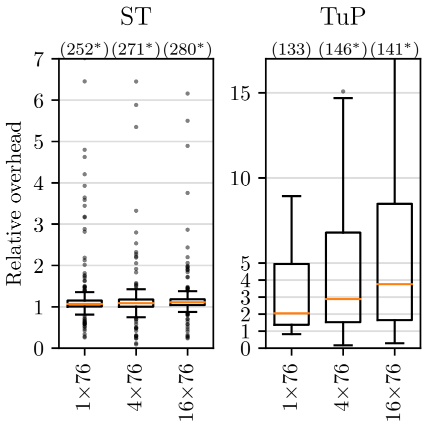
Monolithic proofs [Mic+23]



Overhead relative to solving time w/o LRAT outputs · ST: Solving time · TuP: Time until Proof present · TuV: Time until Validation done
*some data outside of displayed domain

Overhead Relative to Proof-free Solving

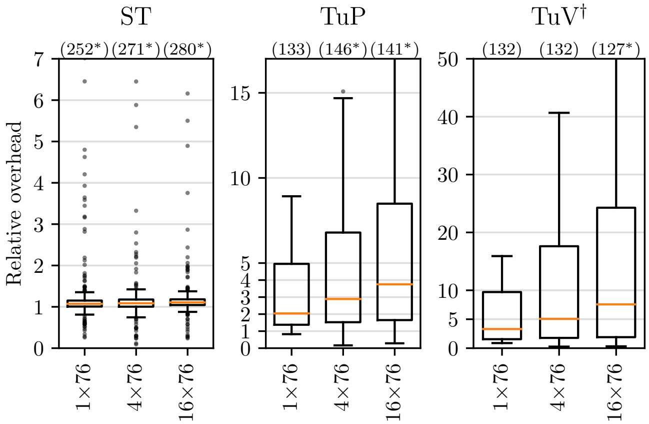
Monolithic proofs [Mic+23]



Overhead relative to solving time w/o LRAT outputs · ST: Solving time · TuP: Time until Proof present · TuV: Time until Validation done
*some data outside of displayed domain

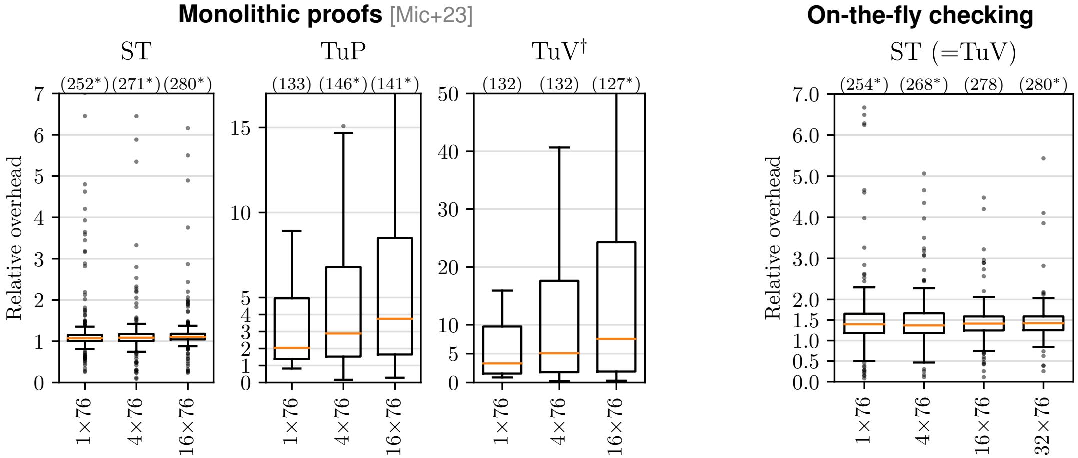
Overhead Relative to Proof-free Solving

Monolithic proofs [Mic+23]



Overhead relative to solving time w/o LRAT outputs · ST: Solving time · TuP: Time until Proof present · TuV: Time until Validation done
*some data outside of displayed domain · †Data extrapolated

Overhead Relative to Proof-free Solving



Overhead relative to solving time w/o LRAT outputs · ST: Solving time · TuP: Time until Proof present · TuV: Time until Validation done
* some data outside of displayed domain · † Data extrapolated

Discussion

Karlsruhe Institute of Technology

- ✓ Generic framework
  - Requires LRAT-producing solver backends
  - Independent of structure, implementation of clause exchange
- ✓ Extended to checking satisfying assignments
  - One checker per solver process needs to remember all original problem clauses

Discussion

Karlsruhe Institute of Technology

- ✓ Generic framework
  - Requires LRAT-producing solver backends
  - Independent of structure, implementation of clause exchange
- ✓ Extended to checking satisfying assignments
  - One checker per solver process needs to remember all original problem clauses
- ✓ Works with malleable scheduling, i.e., with fluctuating set of workers

Discussion

- ✓ Generic framework
  - Requires LRAT-producing solver backends
  - Independent of structure, implementation of clause exchange
- ✓ Extended to checking satisfying assignments
  - One checker per solver process needs to remember all original problem clauses
- ✓ Works with malleable scheduling, i.e., with fluctuating set of workers
- ! High memory usage (+60% compared to proof-less solving)
  - Compressing clauses in checkers?
  - Parallel checking code with shared clause database?

Discussion

✓ Generic framework
  - Requires LRAT-producing solver backends
  - Independent of structure, implementation of clause exchange

✓ Extended to checking satisfying assignments
  - One checker per solver process needs to remember all original problem clauses

✓ Works with malleable scheduling, i.e., with fluctuating set of workers

! High memory usage (+60% compared to proof-less solving)
  - Compressing clauses in checkers?
  - Parallel checking code with shared clause database?

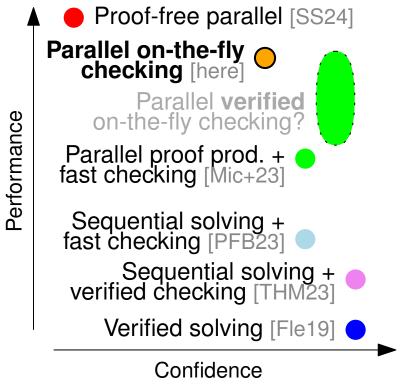
? Formal verification of trusted processes?

Cooperation wanted!

  - Would result in first verified distributed SAT solver (in terms of correctness, not termination)
  - Extend projects like `cake_lpr` [THM23]? Efficient enough?
  - Verify (parts of) C99 codebase? BMC? Verified compilation?

Conclusion

- Bottleneck-free approach to on-the-fly proof checking for distributed clause-sharing solving
- Trusted parties: Isolated parser and checker processes, extending usual LRAT checking interface
- Saves an order of magnitude in running time overhead over explicit proof production
- Paves the road to verified distributed SAT solving



github.com/domschrei/impcheck

References

[AB12] Jean-Philippe Aumasson and Daniel J. Bernstein. "SipHash: a fast short-input PRF". In: *International Conference on Cryptology in India*. Springer. 2012, pp. 489–508. DOI: 10.1007/978-3-642-34931-7_28.

[FB22] Mathias Fleury and Armin Biere. "Scalable Proof Producing Multi-Threaded SAT Solving with Gimsatul through Sharing instead of Copying Clauses". In: *Pragmatics of SAT*. 2022.

[Fle19] Mathias Fleury. "Optimizing a verified SAT solver". In: *NASA Formal Methods: 11th International Symposium, NFM 2019, Houston, TX, USA, May 7–9, 2019, Proceedings 11*. Springer. 2019, pp. 148–165.

[HMP14] Marijn J. H. Heule, Norbert Manthey, and Tobias Philipp. "Validating Unsatisfiability Results of Clause Sharing Parallel SAT Solvers.". In: *Pragmatics of SAT*. 2014, pp. 12–25. DOI: 10.29007/6vwg.

[Mic+23] Dawn Michaelson et al. "Unsatisfiability proofs for distributed clause-sharing SAT solvers". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. 2023, pp. 348–366. DOI: 10.1007/978-3-031-30823-9_18.

[PFB23] Florian Pollitt, Mathias Fleury, and Armin Biere. "Faster LRAT checking than solving with CaDiCaL". In: *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. DOI: 10.4230/LIPIcs.SAT.2023.21.

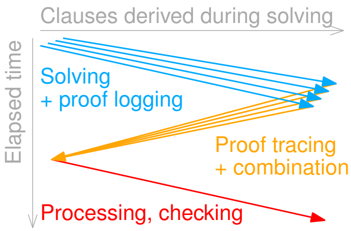
[SS24] Dominik Schreiber and Peter Sanders. "MallobSat: Scalable SAT Solving by Clause Sharing". In: *Journal of Artificial Intelligence Research (JAIR)* (2024). In press.

[THM23] Yong Kiam Tan, Marijn J. H. Heule, and Magnus Myreen. "Verified LRAT and LPR Proof Checking with cake_lpr". In: *SAT Competition*. 2023, p. 89. URL: https://researchportal.helsinki.fi/files/269128852/sc2023_proceedings.pdf.

Intrinsic Scalability Issues

Bottleneck: sequential assembly and checking of monolithic proof

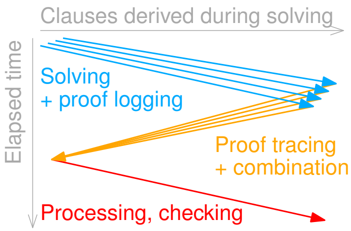
- Throttled by I/O bandwidth at final process
- Sometimes hundreds of Gigabytes of proof information
- Proof production + checking @ 1520 cores takes $\approx 3\times$ solving time
  (latest setup – submitted to JAR)
- Intuition "*If solving fits into RAM, checking will as well*" no longer holds



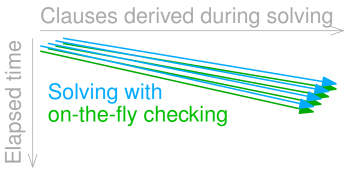
Intrinsic Scalability Issues

Bottleneck: sequential assembly and checking of monolithic proof

- Throttled by I/O bandwidth at final process
- Sometimes hundreds of Gigabytes of proof information
- Proof production + checking @ 1520 cores takes $\approx 3\times$ solving time
  (latest setup – submitted to JAR)
- Intuition "*If solving fits into RAM, checking will as well*" no longer holds



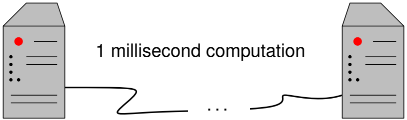
Our aim: Make checking scalable by dropping requirement of a single, persistent proof



The (Un)Likelihood of 2^{-128}

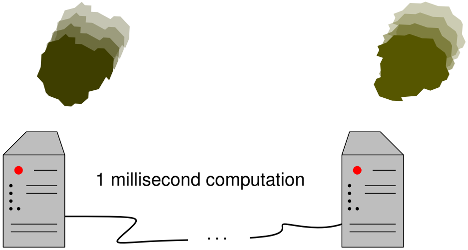
- Estimated (2007) probability of dying due to a **local** comet/asteroid impact: 1 in 5 700 000[1]

  [1] http://www.boulder.swri.edu/clark/binhaz07.ppt
- Average human life span estimate (conservative): 80 years
- Probability of such an impact per millisecond: 1 in $5\,700\,000 \cdot (80 \cdot 365 \cdot 24 \cdot 3600 \cdot 1000) \approx 1.4 \cdot 10^{-19}$
- Two unrelated impacts in the same millisecond: $10^{-19} \cdot 10^{-19} = 10^{-38} \approx 2^{-128}$



The (Un)Likelihood of 2^{-128}

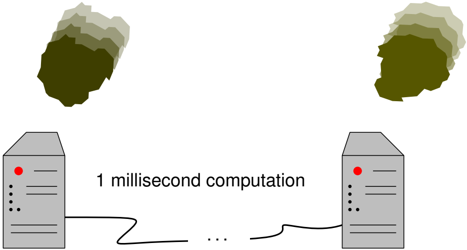
- Estimated (2007) probability of dying due to a **local** comet/asteroid impact: 1 in 5 700 000[1]
  [1] http://www.boulder.swri.edu/clark/binhaz07.ppt
- Average human life span estimate (conservative): 80 years
- Probability of such an impact per millisecond: 1 in $5\,700\,000 \cdot (80 \cdot 365 \cdot 24 \cdot 3600 \cdot 1000) \approx 1.4 \cdot 10^{-19}$
- Two unrelated impacts in the same millisecond: $10^{-19} \cdot 10^{-19} = 10^{-38} \approx 2^{-128}$



The (Un)Likelihood of 2^{-128}

- Estimated (2007) probability of dying due to a **local** comet/asteroid impact: 1 in 5 700 000[1]

  [1] http://www.boulder.swri.edu/clark/binhaz07.ppt
- Average human life span estimate (conservative): 80 years
- Probability of such an impact per millisecond: 1 in $5\,700\,000 \cdot (80 \cdot 365 \cdot 24 \cdot 3600 \cdot 1000) \approx 1.4 \cdot 10^{-19}$
- Two unrelated impacts in the same millisecond: $10^{-19} \cdot 10^{-19} = 10^{-38} \approx 2^{-128}$

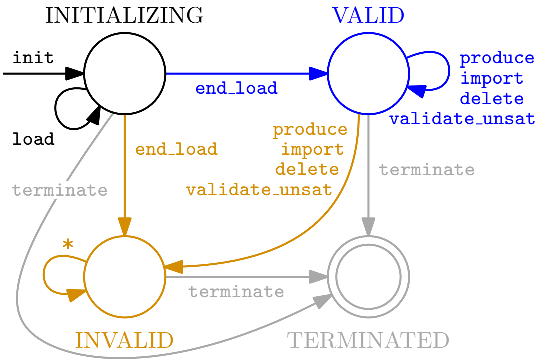


- Same argument with cosmic radiation flipping two particular bytes (prob. 10^{-15} per byte per sec.), causing a formally verified checker to hallucinate unsatisfiability

Checker Interface

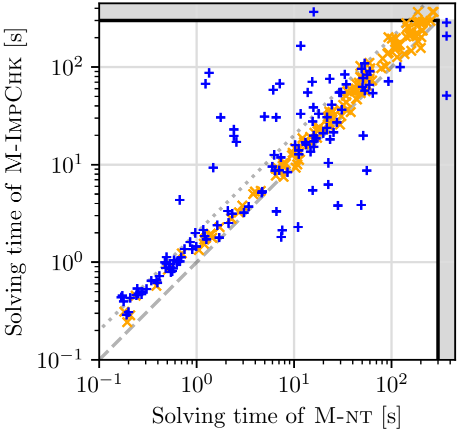
Protocol realized via named pipes:

```
init(sig: Signature) → void
load(formula: ClauseSet) → void
end_load() → bool
produce(id: ID, lits: Clause, hints: IDList, share: bool)
      → (bool, Signature?)
import(id: ID, lits: Clause, sig: Signature) → bool
delete(ids: IDList) → bool
validate_unsat() → (bool, Signature?)
terminate() → void
```

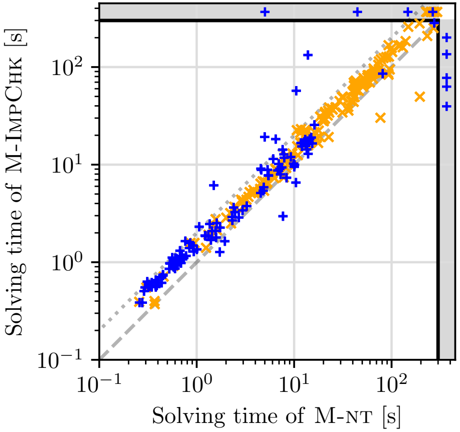


Results: Solving Time Overhead

1 node (76 cores)



32 nodes (2432 cores)



M-nt: MallobSat+CaDiCaL, no LRAT output · M-ImpChk: MallobSat+CaDiCaL + on-the-fly checking

Results: Solving Times (w/o Assembly, Checking)