# Machine Learning based SAT Solvers for Cryptanalysis

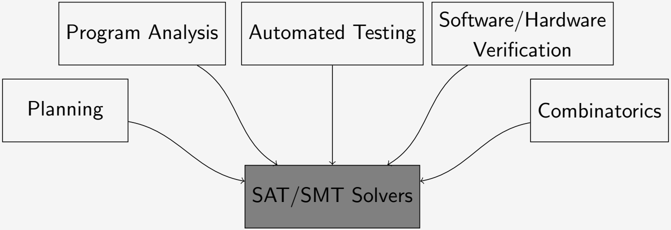
Saeed Nejati

UNIVERSITY OF WATERLOO

April 2nd, 2020

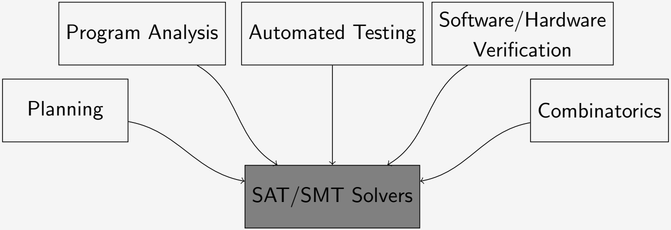
# Motivation

- **SAT Solvers**: Powerful general purpose search tools



# Motivation

- **SAT Solvers**: Powerful general purpose search tools
- **Cryptanalysis**: Searching a huge search space for a secret key/value



# Motivation

- SAT/SMT solvers have increasingly been used in Cryptographic tasks
  - Finding cryptographic keys [Mas99, MM00]
  - Modular root finding [FMM03]
  - A collision attack [MZ06]
  - Preimage attacks [MS13], [Nos12]
  - Differential cryptanalysis [Pro16]
  - RX-differentials [Ashur2017], [DW17]
  - Verification of cryptographic primitives [Tom15]

# Motivation

- SAT/SMT solvers have increasingly been used in Cryptographic tasks
  - Finding cryptographic keys [Mas99, MM00]
  - Modular root finding [FMM03]
  - A collision attack [MZ06]
  - Preimage attacks [MS13], [Nos12]
  - Differential cryptanalysis [Pro16]
  - RX-differentials [Ashur2017], [DW17]
  - Verification of cryptographic primitives [Tom15]
- However, they mostly used SAT solvers as a black-box

# Motivation

- SAT/SMT solvers have increasingly been used in Cryptographic tasks
  - Finding cryptographic keys [Mas99, MM00]
  - Modular root finding [FMM03]
  - A collision attack [MZ06]
  - Preimage attacks [MS13], [Nos12]
  - Differential cryptanalysis [Pro16]
  - RX-differentials [Ashur2017], [DW17]
  - Verification of cryptographic primitives [Tom15]
- However, they mostly used SAT solvers as a black-box

**Question**

Can we use SAT solvers in a white-box fashion?

# Motivation

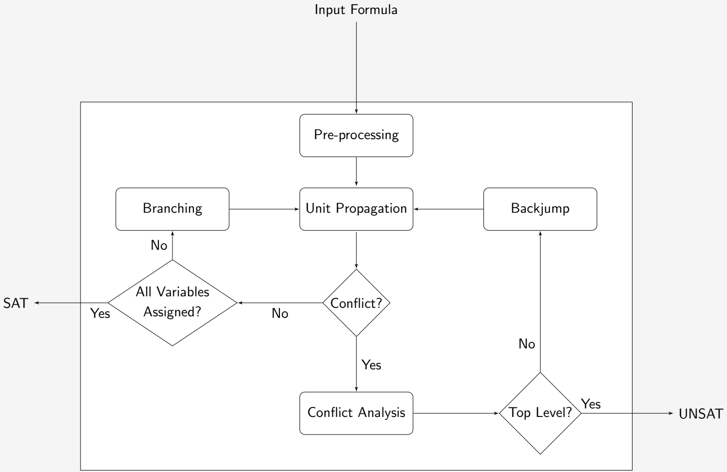
- SAT/SMT solvers have increasingly been used in Cryptographic tasks
  - Finding cryptographic keys [Mas99, MM00]
  - Modular root finding [FMM03]
  - A collision attack [MZ06]
  - Preimage attacks [MS13], [Nos12]
  - Differential cryptanalysis [Pro16]
  - RX-differentials [Ashur2017], [DW17]
  - Verification of cryptographic primitives [Tom15]
- However, they mostly used SAT solvers as a black-box

**Question**

Can we use SAT solvers in a white-box fashion?

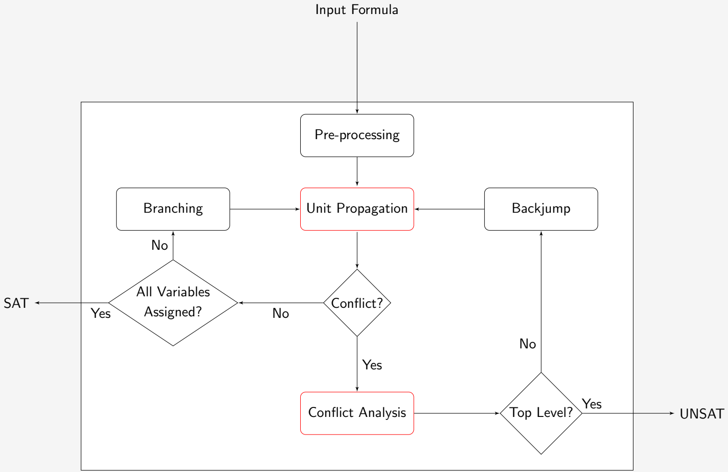
(Tailor internals for a specific cryptographic problem)

# Opening up a SAT solver



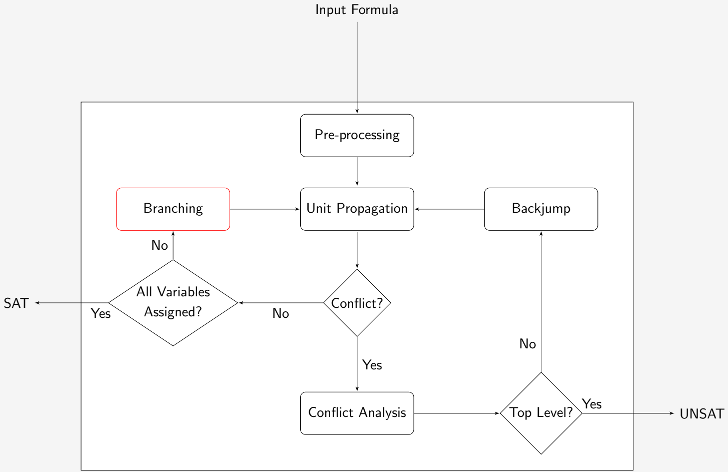

[LG+18]

# Opening up a SAT solver



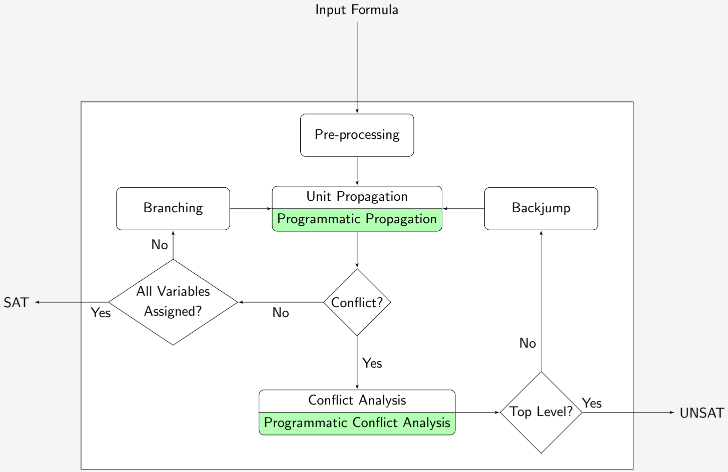

[LG+18]

# Opening up a SAT solver

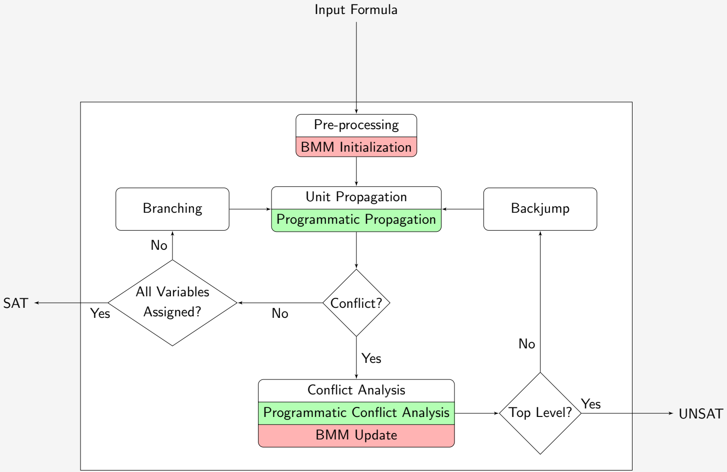



[LG+18]

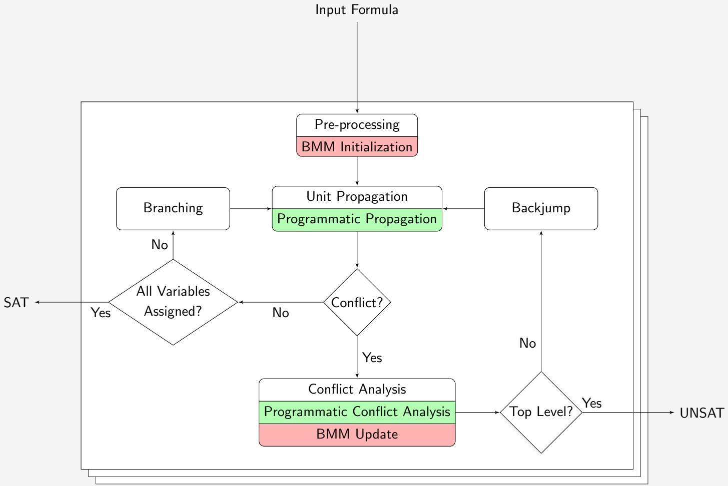
# Opening up a SAT solver



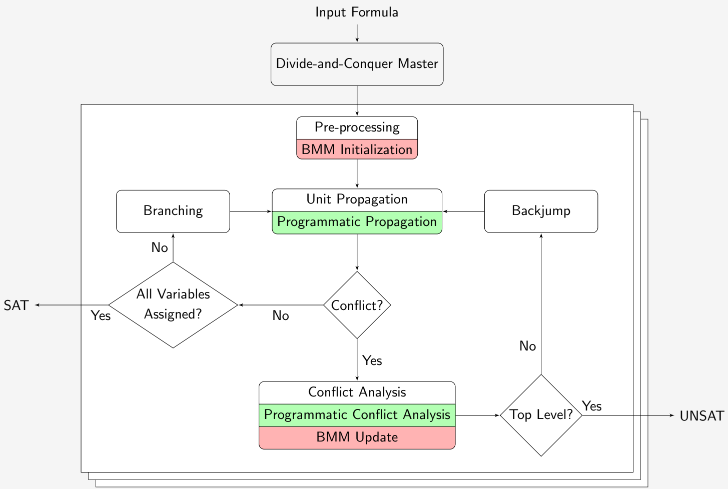
# Opening up a SAT solver



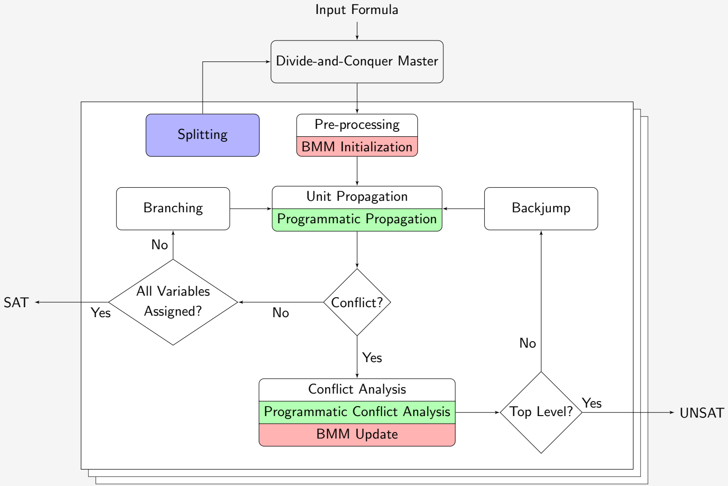
# Opening up a SAT solver



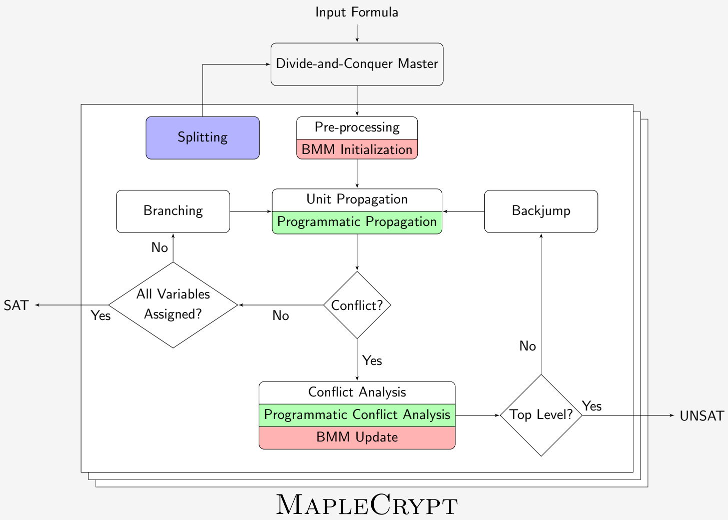
# Opening up a SAT solver



# Opening up a SAT solver



# Opening up a SAT solver

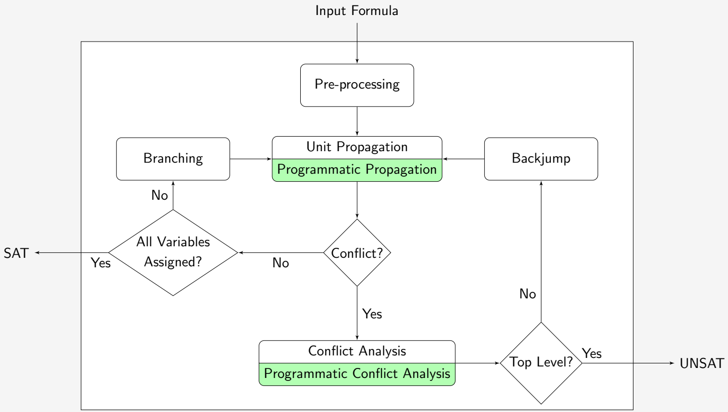


# Outline of Contributions

1. Extending reasoning components for cryptographic problems
   - CDCL(Crypto) framework ([NG19])
   - Algebraic fault attack ([NHGG18])
   - Differential cryptanalysis ([NG19])
2. Improving search heuristics
   - Machine learning for search heuristics optimization problems
   - Sequencing: Splitting heuristics ([NLFG20, NNS+17])
   - Initializing: Variable order and value selection (Branching heuristics) ([NDT+20])

# Part 1: CDCL(Crypto) Solvers

# Overview




CDCL(Crypto): CDCL SAT solver with custom cryptographic reasoning

# Lost in Translation

- When encoding a constraint into SAT, some higher level properties might be lost

- Example: consider a pseudo-Boolean constraint $C : x + y \leq 0, (x, y \in \{0, 1\})$
  - We trivially know: $C \rightarrow \bar{x}$ and $C \rightarrow \bar{y}$.

# Lost in Translation

- When encoding a constraint into SAT, some higher level properties might be lost
- Example: consider a pseudo-Boolean constraint $C : x + y \leq 0, (x, y \in \{0, 1\})$
  - We trivially know: $C \rightarrow \bar{x}$ and $C \rightarrow \bar{y}$.
  - We can encode it using a half-adder

## Lost in Translation

- When encoding a constraint into SAT, some higher level properties might be lost
- Example: consider a pseudo-Boolean constraint $C : x + y \leq 0, (x, y \in \{0, 1\})$
  - We trivially know: $C \rightarrow \bar{x}$ and $C \rightarrow \bar{y}$.
  - We can encode it using a half-adder
  - $sum \leftrightarrow x \oplus y$, $carry \leftrightarrow x \wedge y$, and adding constraints $sum = 0, carry = 0$.

# Lost in Translation

- When encoding a constraint into SAT, some higher level properties might be lost
- Example: consider a pseudo-Boolean constraint $C : x + y \leq 0, (x, y \in \{0, 1\})$
  - We trivially know: $C \rightarrow \bar{x}$ and $C \rightarrow \bar{y}$.
  - We can encode it using a half-adder
  - $sum \leftrightarrow x \oplus y$, $carry \leftrightarrow x \wedge y$, and adding constraints $sum = 0, carry = 0$.
  - Resultant CNF: $(\neg x \vee \neg y) \wedge (\neg x \vee y) \wedge (x \vee y)$

# Lost in Translation

- When encoding a constraint into SAT, some higher level properties might be lost

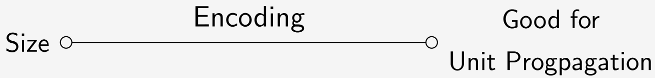
- Example: consider a pseudo-Boolean constraint $C : x + y \leq 0, (x, y \in \{0, 1\})$
    - We trivially know: $C \to \bar{x}$ and $C \to \bar{y}$.
    - We can encode it using a half-adder
    - $sum \leftrightarrow x \oplus y$, $carry \leftrightarrow x \wedge y$, and adding constraints $sum = 0, carry = 0$.
    - Resultant CNF: $(\neg x \vee \neg y) \wedge (\neg x \vee y) \wedge (x \vee y)$
    - No unit clause to propagate!

# Encoding and Propagation

Encoding

Size ——— Good for Unit Progpagation

# Encoding and Propagation




- Ideal: Having "good" propagation while keeping the encoding small

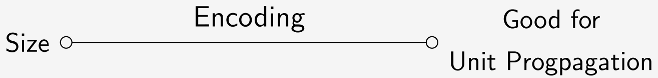
# Encoding and Propagation

Encoding

Size — Good for Unit Progpagation

- Ideal: Having "good" propagation while keeping the encoding small
- Extending propagation programmatically

# Encoding and Propagation




- Ideal: Having “good” propagation while keeping the encoding small
- Extending propagation programmatically
- Using Programmatic SAT architecture [GOS+12]

# Programmatic SAT

- Instrumenting a SAT solver with *callbacks*

# Programmatic SAT

- Instrumenting a SAT solver with *callbacks*
- Extending functionality of propagation and conflict analysis

# Programmatic SAT

- Instrumenting a SAT solver with *callbacks*
- Extending functionality of propagation and conflict analysis
- Similar to and derived from CDCL($T$) paradigm [NOT06]

# Programmatic SAT

- Instrumenting a SAT solver with *callbacks*
- Extending functionality of propagation and conflict analysis
- Similar to and derived from CDCL($T$) paradigm [NOT06]
- Programmatic callbacks analyze the partial assignment

# Programmatic SAT

- Instrumenting a SAT solver with *callbacks*
- Extending functionality of propagation and conflict analysis
- Similar to and derived from CDCL($T$) paradigm [NOT06]
- Programmatic callbacks analyze the partial assignment
- Propagation callback
  - Called after unit propagation
  - Checks for implied literals that are missed by unit propagation

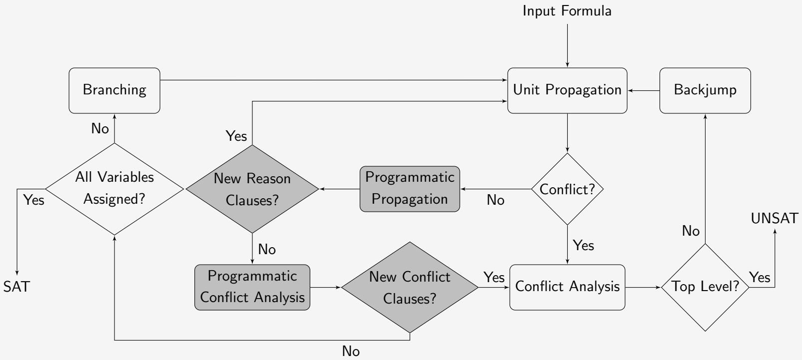
## Programmatic SAT

- Instrumenting a SAT solver with *callbacks*
- Extending functionality of propagation and conflict analysis
- Similar to and derived from CDCL($T$) paradigm [NOT06]
- Programmatic callbacks analyze the partial assignment
- Propagation callback
  - Called after unit propagation
  - Checks for implied literals that are missed by unit propagation
- Conflict analysis callback
  - Called after *propagation* is done
  - Checks if partial assignment cannot be extended to a full solution

## Programmatic SAT

- Instrumenting a SAT solver with *callbacks*
- Extending functionality of propagation and conflict analysis
- Similar to and derived from CDCL($T$) paradigm [NOT06]
- Programmatic callbacks analyze the partial assignment
- Propagation callback
    - Called after unit propagation
    - Checks for implied literals that are missed by unit propagation
- Conflict analysis callback
    - Called after *propagation* is done
    - Checks if partial assignment cannot be extended to a full solution
- It can be seen as as solver for hybrid "CNF+C" constraints.

# Programmatic SAT



## Case Studies

- Applied this framework to two cryptographic problems:
  - Algebraic Fault Attack on SHA-1 and SHA-256
  - Differential Cryptanalysis of round-reduced version of SHA-256

# Algebraic Fault Analysis

- Implementation attack on a crypto function with an embedded secret

# Algebraic Fault Analysis

- Implementation attack on a crypto function with an embedded secret
- Inducing faults in the process of target function

# Algebraic Fault Analysis

- Implementation attack on a crypto function with an embedded secret
- Inducing faults in the process of target function
- Pre-image: given $H$, find an $m$, s.t. $SHA(m) = H$.

# Algebraic Fault Analysis

- Implementation attack on a crypto function with an embedded secret
- Inducing faults in the process of target function
- Pre-image: given $H$, find an $m$, s.t. $SHA(m) = H$.
- Very hard by itself.

# Algebraic Fault Analysis

- Implementation attack on a crypto function with an embedded secret
- Inducing faults in the process of target function
- Pre-image: given $H$, find an $m$, s.t. $SHA(m) = H$.
- Very hard by itself.
- Collect extra information (constraints) about the secret $m$

# Algebraic Fault Analysis

- Implementation attack on a crypto function with an embedded secret
- Inducing faults in the process of target function
- Pre-image: given $H$, find an $m$, s.t. $SHA(m) = H$.
- Very hard by itself.
- Collect extra information (constraints) about the secret $m$
- Inject fault in a target register: $SHA'(m) = H'$

## Algebraic Fault Analysis

- Implementation attack on a crypto function with an embedded secret
- Inducing faults in the process of target function
- Pre-image: given $H$, find an $m$, s.t. $SHA(m) = H$.
- Very hard by itself.
- Collect extra information (constraints) about the secret $m$
- Inject fault in a target register: $SHA'(m) = H'$
- and repeat $SHA''(m) = H''$

# Algebraic Fault Attack on SHA functions

- SHA functions: Iteratively applying a round function

# Algebraic Fault Attack on SHA functions

- SHA functions: Iteratively applying a round function
- Each round mixes one word of message with state variables

# Algebraic Fault Attack on SHA functions

- SHA functions: Iteratively applying a round function
- Each round mixes one word of message with state variables
- SHA-1$(m)$ : $f_{79} \circ f_{78} \circ \cdots \circ f_1 \circ f_0(m)$

# Algebraic Fault Attack on SHA functions

- SHA functions: Iteratively applying a round function
- Each round mixes one word of message with state variables
- SHA-1$(m)$ : $f_{79} \circ f_{78} \circ \cdots \circ f_1 \circ f_0(m)$
- Slice the function into smaller number of rounds and inject fault in between
- Focus on last 16 rounds

# Algebraic Fault Attack on SHA functions

- SHA functions: Iteratively applying a round function
- Each round mixes one word of message with state variables
- SHA-1$(m)$ : $f_{79} \circ f_{78} \circ \cdots \circ f_1 \circ f_0(m)$
- Slice the function into smaller number of rounds and inject fault in between
- Focus on last 16 rounds
- SHA-1$(m)$ :$f_{79} \circ \cdots \circ f_{64} \circ f_{63} \circ \cdots \circ f_1 \circ f_0(m)$

## Algebraic Fault Attack on SHA functions

- SHA functions: Iteratively applying a round function
- Each round mixes one word of message with state variables
- SHA-1$(m)$ : $f_{79} \circ f_{78} \circ \cdots \circ f_1 \circ f_0(m)$
- Slice the function into smaller number of rounds and inject fault in between
- Focus on last 16 rounds
- SHA-1$(m)$ :$f_{79} \circ \cdots \circ f_{64} \circ f_{63} \circ \cdots \circ f_1 \circ f_0(m)$
- Model fault injection with a random value
- $H'_i = f_{64..79}(f_{0..63}(m_{0..63}) \oplus \delta_i, m_{64..79})$

## Algebraic Fault Attack on SHA functions

- SHA functions: Iteratively applying a round function
- Each round mixes one word of message with state variables
- SHA-1$(m)$ : $f_{79} \circ f_{78} \circ \cdots \circ f_1 \circ f_0(m)$
- Slice the function into smaller number of rounds and inject fault in between
- Focus on last 16 rounds
- SHA-1$(m)$ : $f_{79} \circ \cdots \circ f_{64} \circ f_{63} \circ \cdots \circ f_1 \circ f_0(m)$
- Model fault injection with a random value
- $H'_i = f_{64..79}(\underbrace{f_{0..63}(m_{0..63})} \oplus \delta_i, m_{64..79})$
- Unaffected parts are just repeated.

# Algebraic Fault Attack on SHA functions

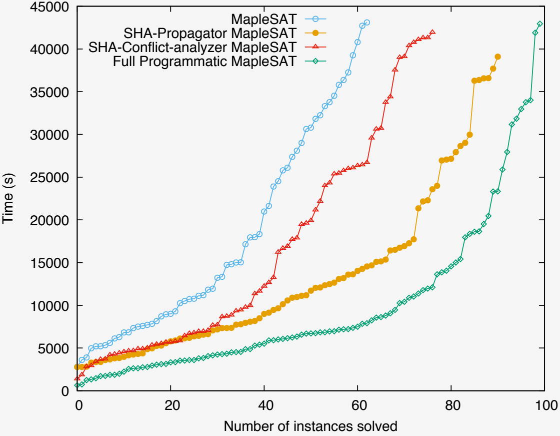
- SHA functions: Iteratively applying a round function
- Each round mixes one word of message with state variables
- SHA-1$(m) : f_{79} \circ f_{78} \circ \cdots \circ f_1 \circ f_0(m)$
- Slice the function into smaller number of rounds and inject fault in between
- Focus on last 16 rounds
- SHA-1$(m) : f_{79} \circ \cdots \circ f_{64} \circ f_{63} \circ \cdots \circ f_1 \circ f_0(m)$
- Model fault injection with a random value
- $H'_i = f_{64..79}(\underbrace{f_{0..63}(m_{0..63})} \oplus \delta_i, m_{64..79})$
- Unaffected parts are just repeated. Abstract them away.

# Algebraic Fault Analysis - Programmatic Approach

- Base SAT solver: MapleSAT
- Programmatic conflict analyzer
  - Embedding the verification loop
  - As soon as message word variables are set, they are ready to be verified
  - Early embedded check vs. Straightforward check after solving completely
- Programmatic propagator
  - Improving the propagation flow of multi-operand additions
  - Generating *reason clauses* in each column addition when output bits are missed

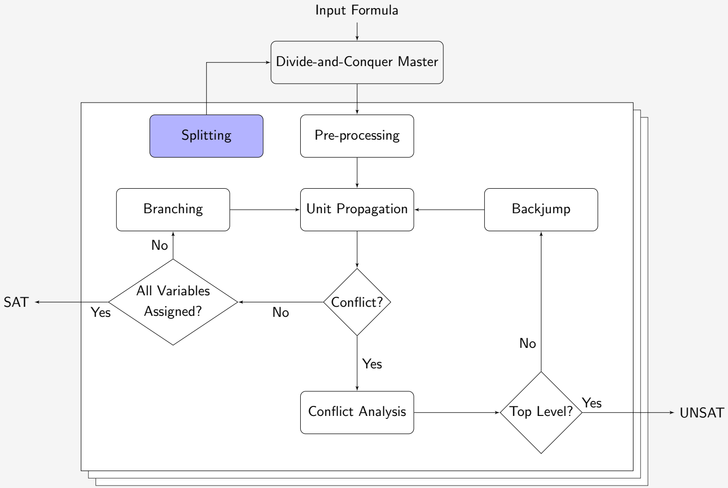
# Algebraic Fault Analysis - Results

- Recovering SHA-256 message bits
- 14.3x speed-up on average
- 17 fewer faults were needed compared to the previous works



# Part 2: Machine Learning based Splitting Heuristics in Parallel SAT Solvers

# Overview


Input Formula
Divide-and-Conquer Master
Splitting
Pre-processing
Branching
Unit Propagation
Backjump
No
All Variables Assigned?
Conflict?
SAT
Yes
No
Yes
No
Conflict Analysis
Top Level?
Yes
UNSAT

# Parallel SAT Solvers

Divide-and-Conquer Solvers

- Split the formula into several sub-formulas and solve them in parallel

# Parallel SAT Solvers

Divide-and-Conquer Solvers

- Split the formula into several sub-formulas and solve them in parallel
- Solvers share information

# Parallel SAT Solvers

Divide-and-Conquer Solvers

- Split the formula into several sub-formulas and solve them in parallel
- Solvers share information
- Splitting the formula $\phi$:

## Parallel SAT Solvers

Divide-and-Conquer Solvers

- Split the formula into several sub-formulas and solve them in parallel
- Solvers share information
- Splitting the formula $\phi$:
  - Pick a variable $x$ in $\phi$

# Parallel SAT Solvers

Divide-and-Conquer Solvers

- Split the formula into several sub-formulas and solve them in parallel
- Solvers share information
- Splitting the formula $\phi$:
  - Pick a variable $x$ in $\phi$
  - Generate two sub-formulas $\phi_1 = \phi[\neg x]$ and $\phi_2 = \phi[x]$

# Parallel SAT Solvers

Divide-and-Conquer Solvers

- Split the formula into several sub-formulas and solve them in parallel
- Solvers share information
- Splitting the formula $\phi$:
    - Pick a variable $x$ in $\phi$
    - Generate two sub-formulas $\phi_1 = \phi[\neg x]$ and $\phi_2 = \phi[x]$
    - Repeat for $\phi_1$ and $\phi_2$

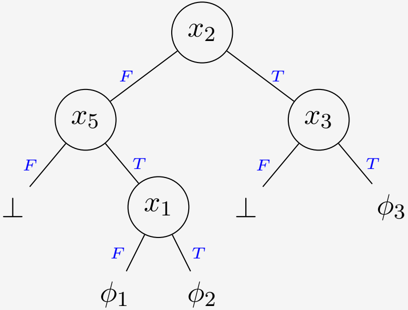
# Parallel SAT Solvers

Divide-and-Conquer Solvers

- Split the formula into several sub-formulas and solve them in parallel
- Solvers share information
- Splitting the formula $\phi$:
  - Pick a variable $x$ in $\phi$
  - Generate two sub-formulas $\phi_1 = \phi[\neg x]$ and $\phi_2 = \phi[x]$
  - Repeat for $\phi_1$ and $\phi_2$
- $\phi$ is SAT: At least one solver returns SAT
- $\phi$ is UNSAT: All solvers return UNSAT

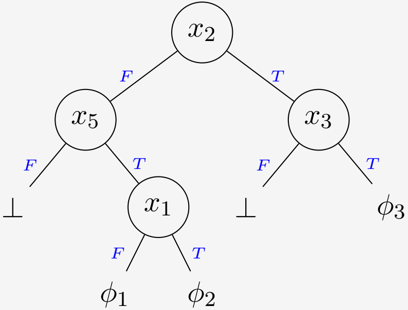
# Search Space Splitting

- $\phi_1 = \phi \wedge \neg x_2 \wedge x_5 \wedge \neg x_1$
- $\phi_2 = \phi \wedge \neg x_2 \wedge x_5 \wedge x_1$
- $\phi_3 = \phi \wedge x_2 \wedge x_3$



# Search Space Splitting

- $\phi_1 = \phi \wedge \neg x_2 \wedge x_5 \wedge \neg x_1$
- $\phi_2 = \phi \wedge \neg x_2 \wedge x_5 \wedge x_1$
- $\phi_3 = \phi \wedge x_2 \wedge x_3$




**Question (Splitting Heuristic)**

How to "divide" so the "conquer" becomes easier?

## Performance Metric

- Q: How do we know a splitting variable is good?

# Performance Metric

- Q: How do we know a splitting variable is good?
- We need to quantify the quality of a splitting variable.

## Performance Metric

- Q: How do we know a splitting variable is good?
- We need to quantify the quality of a splitting variable.
- Performance metric: $pm : \phi \times v \to \mathbb{R}$

## Performance Metric

- Q: How do we know a splitting variable is good?
- We need to quantify the quality of a splitting variable.
- Performance metric: $pm : \phi \times v \rightarrow \mathbb{R}$
- $SplittingHeuristic(\phi) = argmin_{v \in vars(\phi)}\{pm(\phi, v)\}$

## Performance Metric

- Q: How do we know a splitting variable is good?
- We need to quantify the quality of a splitting variable.
- Performance metric: $pm : \phi \times v \to \mathbb{R}$
- $SplittingHeuristic(\phi) = argmin_{v \in vars(\phi)}\{pm(\phi, v)\}$
- The ultimate goal is to minimize the runtime.

## Performance Metric

- Q: How do we know a splitting variable is good?
- We need to quantify the quality of a splitting variable.
- Performance metric: $pm : \phi \times v \rightarrow \mathbb{R}$
- $SplittingHeuristic(\phi) = argmin_{v \in vars(\phi)}\{pm(\phi, v)\}$
- The ultimate goal is to minimize the runtime.
- We define $pm(\phi, v)$: Total wall-clock runtime of solving $\phi$ when splitting once and solving $\phi[v]$ and $\phi[\neg v]$ in parallel.

# Building the Splitting Heuristic

- Computing this $pm$ needs knowing the runtime and status of sub-formulas
- We don't know the runtime a priori
- We can build a machine learning model to predict runtime
- Predicting runtime is a very challenging task

# Building the Splitting Heuristic

- Computing this $pm$ needs knowing the runtime and status of sub-formulas
- We don't know the runtime a priori
- We can build a machine learning model to predict runtime
- Predicting runtime is a very challenging task
- Observation: We are looking for a minimum element in a list of elements ordered by $pm$

## Learn to Rank

- Instead of predicting $pm$ values for each item

# Learn to Rank

- Instead of predicting $pm$ values for each item
- Predict how they compare to each other

# Learn to Rank

- Instead of predicting $pm$ values for each item
- Predict how they compare to each other
- This predictor can be used as a comparator to find the minimum

## Learn to Rank

- Instead of predicting $pm$ values for each item
- Predict how they compare to each other
- This predictor can be used as a comparator to find the minimum
- Goal: given two variables $v$ and $u$ in formula $\phi$:
  - Q: is $v$ better than $u$ for splitting $\phi$?

## Learn to Rank

- Instead of predicting $pm$ values for each item
- Predict how they compare to each other
- This predictor can be used as a comparator to find the minimum
- Goal: given two variables $v$ and $u$ in formula $\phi$:
  - Q: is $v$ better than $u$ for splitting $\phi$?

$$PW(\phi, v_i, v_j) = \begin{cases} 1, & pm(\phi, v_i) < pm(\phi, v_j) \\ 0, & otherwise \end{cases}$$

# Learning $PW$

$$\langle F_{feat}(\phi), V_{feat}(v_i), V_{feat}(v_j), label : (pm(\phi, v_i) < pm(\phi, v_j))\rangle$$

# Learning $PW$

$$\langle F_{feat}(\phi), V_{feat}(v_i), V_{feat}(v_j), label : (pm(\phi, v_i) < pm(\phi, v_j))\rangle$$

- Formula Features:
  - #Variables, #Clauses, AvgVariableNodeDegree, $\cdots$

# Learning $PW$

$$\langle F_{feat}(\phi), V_{feat}(v_i), V_{feat}(v_j), label : (pm(\phi, v_i) < pm(\phi, v_j))\rangle$$

- Formula Features:
  - #Variables, #Clauses, AvgVariableNodeDegree, $\cdots$
- Variable Features:
  - #inBinaryClause, #inTernaryClause, $\cdots$
  - CombinedLRB, PropagationRate, #Flips, $\cdots$

## Learning $PW$

$$\langle F_{feat}(\phi), V_{feat}(v_i), V_{feat}(v_j), label : (pm(\phi, v_i) < pm(\phi, v_j))\rangle$$

- Formula Features:
  - #Variables, #Clauses, AvgVariableNodeDegree, $\cdots$
- Variable Features:
  - #inBinaryClause, #inTernaryClause, $\cdots$
  - CombinedLRB, PropagationRate, #Flips, $\cdots$
- Feature selection:
  - Addition pass: sorted by importance
  - Deletion pass: sorted by computation time

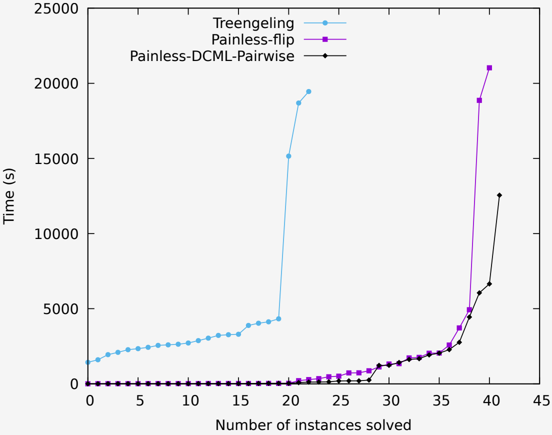
## Learning $PW$

$$\langle F_{feat}(\phi), V_{feat}(v_i), V_{feat}(v_j), label : (pm(\phi, v_i) < pm(\phi, v_j))\rangle$$

- Formula Features:
  - #Variables, #Clauses, AvgVariableNodeDegree, $\cdots$
- Variable Features:
  - #inBinaryClause, #inTernaryClause, $\cdots$
  - CombinedLRB, PropagationRate, #Flips, $\cdots$
- Feature selection:
  - Addition pass: sorted by importance
  - Deletion pass: sorted by computation time
- Random Forest: accuracy 80.72%

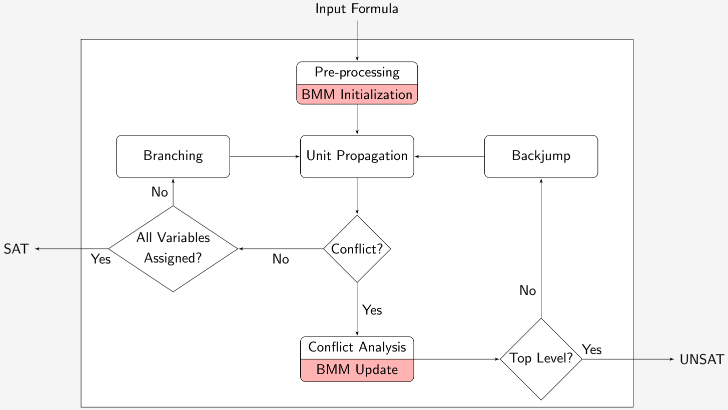
# Experimental Results - Cryptographic benchmark

- Framework: Painless
- Baseline: Painless-DC w/ flip splitting heuristic
- SHA-1 preimage



# Part 3: BMM-based Heuristic Initialization

# Overview



Input Formula
Pre-processing
BMM Initialization
Branching
Unit Propagation
Backjump
No
All Variables Assigned?
SAT
Yes
No
Conflict?
Yes
No
Conflict Analysis
BMM Update
Top Level?
Yes
UNSAT

## Heuristic Initialization

- Branching heuristics: variable selection and value selection (polarity)

# Heuristic Initialization

- Branching heuristics: variable selection and value selection (polarity)
- Usually *look-back*: make a decision based on the gathered search statistics

# Heuristic Initialization

- Branching heuristics: variable selection and value selection (polarity)
- Usually *look-back*: make a decision based on the gathered search statistics
- At the start of search: no statistics available

# Heuristic Initialization

- Branching heuristics: variable selection and value selection (polarity)
- Usually *look-back*: make a decision based on the gathered search statistics
- At the start of search: no statistics available
- Goal: derive variable score and preferred value initial values, s.t. the runtime is improved.

## Bayesian Moment Matching (BMM) for SAT

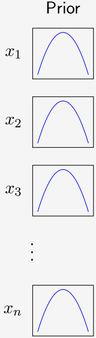
- For each variable: $P(x = T)$: probability of setting $x$ to True

# Bayesian Moment Matching (BMM) for SAT

- For each variable: $P(x = T)$: probability of setting $x$ to True
- Goal: learn a distribution that satisfies all of the clauses

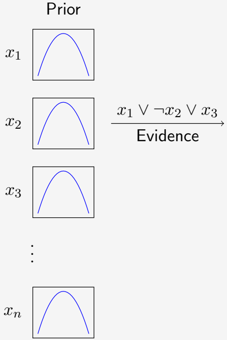
# Bayesian Moment Matching (BMM) for SAT

- For each variable: $P(x = T)$: probability of setting $x$ to True
- Goal: learn a distribution that satisfies all of the clauses



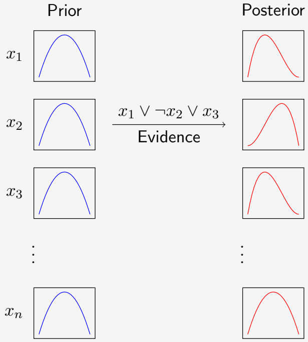
# Bayesian Moment Matching (BMM) for SAT

- For each variable: $P(x = T)$: probability of setting $x$ to True
- Goal: learn a distribution that satisfies all of the clauses



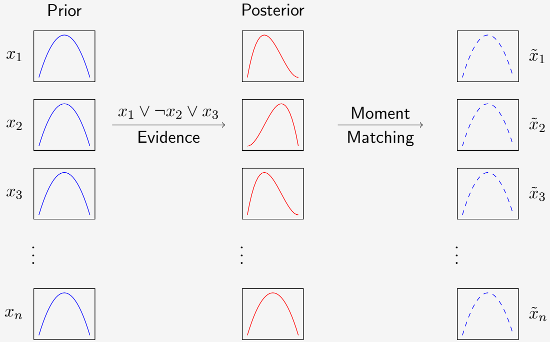
# Bayesian Moment Matching (BMM) for SAT

- For each variable: $P(x = T)$: probability of setting $x$ to True
- Goal: learn a distribution that satisfies all of the clauses



# Bayesian Moment Matching (BMM) for SAT

- For each variable: $P(x = T)$: probability of setting $x$ to True
- Goal: learn a distribution that satisfies all of the clauses

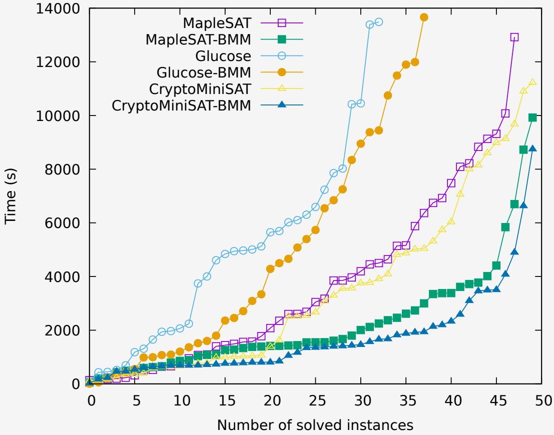


## Heuristic Initialization

- Polarity
  - BMM probabilities collectively represent an assignment
  - $Polarity[x] = \begin{cases} False, & P(x = T) < 0.5 \\ True, & P(x = T) \geq 0.5 \end{cases}$
- Activity
  - Give higher priority to variables that BMM is more *confident* about its polarity
  - $Activity[x] = \begin{cases} 1 - P(x = T), & P(x = T) < 0.5 \\ P(x = T), & P(x = T) \geq 0.5 \end{cases}$

# Experimental Results

- SHA-1 preimage benchmark
- Apple-to-apple comparison
- BMM on MapleSAT, Glucose and CryptoMiniSAT



# Summary and Takeaways

**Extending Reasoning Components**

**Improving Search Heuristics**

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

**Extending Reasoning Components**

Programmatic SAT

**Improving Search Heuristics**

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

**Extending Reasoning Components**

CDCL(Crypto) framework

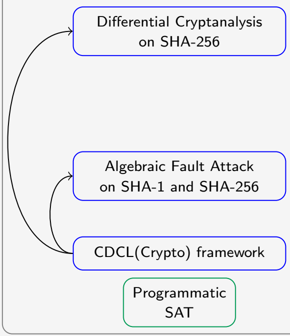
Programmatic SAT

**Improving Search Heuristics**

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

### Extending Reasoning Components




### Improving Search Heuristics

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

**Extending Reasoning Components**

Differential Cryptanalysis
on SHA-256

Recovering secret with 17 fewer faults
14.3x speedup

Algebraic Fault Attack
on SHA-1 and SHA-256

CDCL(Crypto) framework

Programmatic
SAT

**Improving Search Heuristics**

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

### Extending Reasoning Components

SAT-based SHA-256 collision: 1 more round

Differential Cryptanalysis
on SHA-256

Recovering secret with 17 fewer faults
14.3x speedup

Algebraic Fault Attack
on SHA-1 and SHA-256

CDCL(Crypto) framework

Programmatic
SAT

### Improving Search Heuristics

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

### Extending Reasoning Components

SAT-based SHA-256 collision: 1 more round

Differential Cryptanalysis
on SHA-256

Recovering secret with 17 fewer faults
14.3x speedup

Algebraic Fault Attack
on SHA-1 and SHA-256

CDCL(Crypto) framework

Programmatic
SAT

### Improving Search Heuristics

ML for search heuristics
optimization problems

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

### Extending Reasoning Components

SAT-based SHA-256 collision: 1 more round

Differential Cryptanalysis
on SHA-256

Recovering secret with 17 fewer faults
14.3x speedup

Algebraic Fault Attack
on SHA-1 and SHA-256

CDCL(Crypto) framework

Programmatic
SAT

### Improving Search Heuristics

Sequencing:
Pairwise ranking

ML for search heuristics
optimization problems

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

### Extending Reasoning Components

SAT-based SHA-256 collision: 1 more round

Differential Cryptanalysis
on SHA-256

Recovering secret with 17 fewer faults
14.3x speedup

Algebraic Fault Attack
on SHA-1 and SHA-256

CDCL(Crypto) framework

Programmatic
SAT

### Improving Search Heuristics

Splitting Heuristics

Sequencing:
Pairwise ranking

ML for search heuristics
optimization problems

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

## Extending Reasoning Components

SAT-based SHA-256 collision: 1 more round

Differential Cryptanalysis
on SHA-256

Recovering secret with 17 fewer faults
14.3x speedup

Algebraic Fault Attack
on SHA-1 and SHA-256

CDCL(Crypto) framework

Programmatic
SAT

## Improving Search Heuristics

more instances on
SHA-1 preimage

Splitting Heuristics

Sequencing:
Pairwise ranking

ML for search heuristics
optimization problems

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

### Extending Reasoning Components

SAT-based SHA-256 collision: 1 more round

Differential Cryptanalysis on SHA-256

Recovering secret with 17 fewer faults
14.3x speedup

Algebraic Fault Attack on SHA-1 and SHA-256

CDCL(Crypto) framework

Programmatic SAT

### Improving Search Heuristics

more instances on SHA-1 preimage

Splitting Heuristics

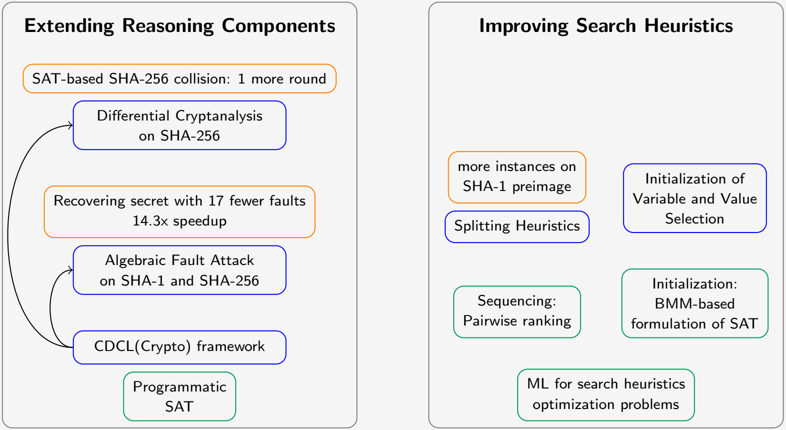
Sequencing: Pairwise ranking

Initialization: BMM-based formulation of SAT

ML for search heuristics optimization problems

- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways




- Key insights from literature
- Our designs
- Our results

# Summary and Takeaways

### Extending Reasoning Components

SAT-based SHA-256 collision: 1 more round

Differential Cryptanalysis on SHA-256

Recovering secret with 17 fewer faults 14.3x speedup

Algebraic Fault Attack on SHA-1 and SHA-256

CDCL(Crypto) framework

Programmatic SAT

### Improving Search Heuristics

2x speedup on SHA-1 preimage

more instances on SHA-1 preimage

Initialization of Variable and Value Selection

Splitting Heuristics

Sequencing: Pairwise ranking

Initialization: BMM-based formulation of SAT

ML for search heuristics optimization problems

- Key insights from literature
- Our designs
- Our results

# Publications

[NLG+17] Nejati, Liang, Gebotys, Czarnecki, Ganesh
*Adaptive restart and CEGAR-based solver for inverting cryptographic hash functions*
**VSTTE 2017**

[NNS+17] Nejati, Newsham, Scott, Liang, Gebotys, Poupart, Ganesh
*A propagation rate based splitting heuristic for divide-and-conquer solvers*
**SAT 2017**

[NHGG18] Nejati, Horáček, Gebotys, Ganesh
*Algebraic fault attack on SHA hash functions using programmatic SAT solvers*
**CP 2018**

[NG19] Nejati, Ganesh
*CDCL(Crypto) SAT solvers for cryptanalysis*
**CASCON 2019**

[NDT+20] Nejati/Duan, Trimponias, Poupart, Ganesh
*Online bayesian moment matching based SAT solver heuristics*
**ICML 2020**

[NLFG20] Nejati, Le Frioux, Ganesh
*A machine learning based splitting heuristic for divide-and-conquer solvers*
**CP 2020**

Thanks!
Questions?

## References i

Glenn De Witte.
**Automatic sat-solver based search tools for cryptanalysis.**

2017.

Claudia Fiorini, Enrico Martinelli, and Fabio Massacci.
**How to Fake an RSA Signature by Encoding Modular Root Finding as a SAT Problem.**
*Discrete Applied Mathematics*, 130(2):101–127, 2003.

Vijay Ganesh, Charles W. O'Donnell, Mate Soos, Srinivas Devadas, Martin C. Rinard, and Armando Solar-Lezama.
**Lynx: A programmatic SAT solver for the RNA-folding problem.**
In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 143–156, 2012.

Fabio Massacci.
**Using Walk-SAT and Rel-SAT for Cryptographic Key Search.**
In *IJCAI*, volume 1999, pages 290–295, 1999.

Fabio Massacci and Laura Marraro.
**Logical Cryptanalysis as a SAT Problem.**
*Journal of Automated Reasoning*, 24(1-2):165–203, 2000.

Paweł Morawiecki and Marian Srebrny.
**A SAT-based Preimage Analysis of Reduced KECCAK Hash Functions.**
*Information Processing Letters*, 113(10):392–397, 2013.

Ilya Mironov and Lintao Zhang.
**Applications of SAT Solvers to Cryptanalysis of Hash Functions.**
*Theory and Applications of Satisfiability Testing-SAT 2006*, pages 102–115, 2006.

Saeed Nejati, Haonan Duan, George Trimponias, Pascal Poupart, and Vijay Ganesh.
**Online bayesian moment matching based sat solver heuristics.**
2020.

Saeed Nejati and Vijay Ganesh.
**Cdcl (crypto) sat solvers for cryptanalysis.**
In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pages 311–316, 2019.

Saeed Nejati, Jan Horáček, Catherine Gebotys, and Vijay Ganesh.
**Algebraic fault attack on sha hash functions using programmatic sat solvers.**
In *International Conference on Principles and Practice of Constraint Programming*, pages 737–754. Springer, 2018.

Saeed Nejati, Ludovic Le Frioux, and Vijay Ganesh.
**A machine learning based splitting heuristic for divide-and-conquer solvers.**
2020.

Saeed Nejati, Jia Hui Liang, Catherine Gebotys, Krzysztof Czarnecki, and Vijay Ganesh.
**Adaptive restart and cegar-based solver for inverting cryptographic hash functions.**
In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 120–131. Springer, 2017.

Saeed Nejati, Zack Newsham, Joseph Scott, Jia Hui Liang, Catherine Gebotys, Pascal Poupart, and Vijay Ganesh.
**A propagation rate based splitting heuristic for divide-and-conquer solvers.**
In *International Conference on Theory and Applications of Satisfiability Testing*, pages 251–260. Springer, 2017.

Vegard Nossum.
**SAT-based Preimage Attacks on SHA-1.**
2012.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli.
**Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t).**
*Journal of the ACM (JACM)*, 53(6):937–977, 2006.

Lukas Prokop.
**Differential cryptanalysis with SAT solvers.**
PhD thesis, University of Technology, Graz, 2016.

Aaron Tomb.
**Applying Satisfiability to the Analysis of Cryptography.**
https://github.com/GaloisInc/sat2015-crypto/blob/master/slides/talk.pdf, 2015.