CDCL(Crypto) and Machine Learning based SAT Solvers for Cryptanalysis

by

Saeed Nejati

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

© Saeed Nejati 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Marijn Heule
Associate Professor
Dept. of Computer Science., Carnegie Mellon University

Supervisor(s): Vijay Ganesh
Associate Professor,
Dept. of Electrical and Computer Eng., University of Waterloo

Catherine Gebotys
Professor,
Dept. of Electrical and Computer Eng., University of Waterloo

Internal Members: Krzysztof Czarnecki
Professor,
Dept. of Electrical and Computer Eng., University of Waterloo

Derek Rayside
Associate Professor,
Dept. of Electrical and Computer Eng., University of Waterloo

Internal-External Member: Pascal Poupart
Professor,
Dept. of Computer Science, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This dissertation includes first-authored peer-reviewed material that has appeared in conference and journal proceedings published by the Springer. The Springer's policy on reuse of published materials in a dissertation is as follows:

> "*Authors have the right to reuse their article's Version of Record, in whole or in part, in their own thesis. Additionally, they may reproduce and make available their thesis, including Springer Nature content, as required by their awarding academic institution. Authors must properly cite the published article in their thesis according to current citation standards.*"

The following list serves as a declaration of the Versions of Record for works included in this dissertation:

Portions of Chapter 3:

- S Nejati, J Horáček, C Gebotys, V Ganesh.
  Algebraic Fault Attack on SHA Hash Functions Using Programmatic SAT Solvers.
  International Conference on Principles and Practice of Constraint Programming, pages 737–754. Springer, Cham, 2018.
- S Nejati, V Ganesh.
  CDCL(Crypto) SAT Solvers for Cryptanalysis.
  Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, pages 311–316. IBM Corp., 2019.

Portions of Chapter 4:

- Unpublished manuscript submitted to ICML 2020:
  S Nejati/H Duan, G Trimponias, P Poupart, V Ganesh.
  Online Bayesian Moment Matching based SAT Solver Heuristics.
  Note: S Nejati and H Duan, were co-first authors of this manuscript.

Portions of Chapter 5:

- S Nejati, Z Newsham, J Scott, JH Liang, C Gebotys, P Poupart, V Ganesh. A propagation rate based splitting heuristic for divide-and-conquer solvers. International Conference on Theory and Applications of Satisfiability Testing, pages 251–260. Springer, Cham, 2017.
- Unpublished manuscript:
  S Nejati, L Le Frioux, V Ganesh.
  A Machine Learning based Splitting Heuristic for Divide-and-Conquer Solvers.

Abstract

Over the last two decades, we have seen a dramatic improvement in the efficiency of conflict-driven clause-learning Boolean satisfiability (CDCL SAT) solvers over industrial problems from a variety of applications such as verification, testing, security, and AI. The availability of such powerful general-purpose search tools as the SAT solver has led many researchers to propose SAT-based methods for cryptanalysis, including techniques for finding collisions in hash functions and breaking symmetric encryption schemes.

A feature of all of the previously proposed SAT-based cryptanalysis work is that they are *blackbox*, in the sense that the cryptanalysis problem is encoded as a SAT instance and then a CDCL SAT solver is invoked to solve said instance. A weakness of this approach is that the encoding thus generated may be too large for any modern solver to solve it efficiently. Perhaps a more important weakness of this approach is that the solver is in no way specialized or tuned to solve the given instance. Finally, very little work has been done to leverage parallelism in the context of SAT-based cryptanalysis.

To address these issues, we developed a set of methods that improve on the state-of-the-art SAT-based cryptanalysis along three fronts. First, we describe an approach called CDCL(Crypto) (inspired by the CDCL(T) paradigm) to tailor the internal subroutines of the CDCL SAT solver with domain-specific knowledge about cryptographic primitives. Specifically, we extend the propagation and conflict analysis subroutines of CDCL solvers with specialized codes that have knowledge about the cryptographic primitive being analyzed by the solver. We demonstrate the power of this framework in two cryptanalysis tasks of algebraic fault attack and differential cryptanalysis of SHA-1 and SHA-256 cryptographic hash functions. Second, we propose a machine-learning based parallel SAT solver that performs well on cryptographic problems relative to many state-of-the-art parallel SAT solvers. Finally, we use a formulation of SAT into Bayesian moment matching to address heuristic initialization problem in SAT solvers.

Acknowledgements

I would like to thank the following people, who made this thesis possible:

My supervisors Vijay Ganesh and Catherine Gebotys. Many thanks to Vijay, for being patient with me and enabling me to mature as a researcher, and guiding me through the minefield of PhD. Thanks to Cathy for all the valuable discussions. Also, I would like to thank Siddharth Garg, who was my supervisor for the first one and a half years, for working with me on interesting hardware security projects.

My colleagues and friends at Computer-Aided Reasoning group, Jimmy, Ed, Curtis, Ian, Joe, Murphy and Hari. Thanks a lot for the insightful discussions and helping me getting through this journey.

My collaborators, Pascal Poupart, Ilias Kotsireas, Jan Horáček and Ludovic Le Frioux, for their help and insight in the research projects that we worked on.

To the examining committee, Marijn Heule, Krzysztof Czarnecki, Derek Rayside and Pascal Poupart, for agreeing to be on my committee and reviewing my thesis and participating in my defense during a pandemic.

My wife, Bahar, without whose support this PhD journey would have not been possible. My friend in PhD, in all the paper deadlines, in conference travels, in internships, in all happy and stressful moments, and in life. I am very happy to have you by my side. I am glad that we both were able to defend our PhD theses despite troubles of COVID-19 pandemic situation.

My amazing parents and sister, who supported me to come to Canada and pursue my PhD, and my extended family who made me feel at home.

My internship mentors, David Tarditi at Microsoft Research, for introducing me to industrial research, and allowing me to explore and learn about Verification and Compiler research in a practical setting. And to Paul Lawrence for a great internship experience at Google.

Dedication

To my wife and my parents.

Table of Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**

1.1 Extending Reasoning Components for Cryptographic Problems 4

1.2 Improving Splitting Heuristics in Parallel SAT Solvers 4

1.3 Initializing SAT Solver Heuristics 5

1.4 Supporting Code Contributions 7

1.5 Supporting Publications 7

2 Background **9**

2.1 Boolean SATisfiability 9

2.2 Conflict-Driven Clause-Learning SAT Solvers 10

2.3 Arc Consistency and SAT 11

2.4 Parallel SAT Solvers 12

2.4.1 Divide-and-Conquer Solvers 13

2.5 Machine Learning 16

2.5.1 Supervised Learning 16

2.5.2 Bayesian Moment Matching and SAT . . . 17
2.6 Cryptographic Hash Functions . . . 20
2.6.1 Description of SHA-1 . . . 21
2.6.2 Description of SHA-256 . . . 23
2.7 Encoding SHA-1 and SHA-256 into SAT . . . 23
2.8 Algebraic Fault Attack . . . 26
2.9 Differential Cryptanalysis . . . 27
2.10 Terminology . . . 29

3 CDCL(Crypto) SAT Solvers 30
3.1 Contributions . . . 31
3.2 CDCL(Crypto) Framework . . . 32
3.2.1 Programmatic Interface in SAT Solvers . . . 32
3.2.2 Cryptographic Reasoning in Programmatic Callbacks . . . 33
3.3 Algebraic Fault Attack . . . 34
3.3.1 High-level Overview of Our Method to Algebraic Fault Attack . . . 35
3.4 Programmatic Callbacks for Algebraic Fault Attack . . . 37
3.4.1 Programmatic Conflict Analysis . . . 37
3.4.2 Programmatic Propagation . . . 37
3.5 Algebraic Fault Attack on SHA-1 and SHA-256 . . . 39
3.5.1 Algebraic Fault Notations . . . 39
3.5.2 Attack Model . . . 39
3.5.3 Attack on SHA-1 . . . 41
3.5.4 Attack on SHA-256 . . . 41
3.6 Experimental Results . . . 44
3.6.1 Experimental Setup . . . 44
3.6.2 Attack on SHA-1 and SHA-256 . . . 44
3.6.3 Performance of the Solver . . . 45

3.6.4 Discussion 47
3.7 Differential Cryptanalysis 48
3.8 Related Work 50
3.9 Chapter Summary 53

4 Initialization of SAT Heuristics 54
4.1 Contributions 55
4.2 Bayesian Moment Matching as a SAT Solver Component 56
4.3 Description of Other Initialization Methods 59
4.4 Experimental Results 60
4.4.1 Evaluation over Hard Cryptographic Instances 60
4.4.2 Evaluation over SAT Competition 2018 and SAT Race 2019 Application Instances 62
4.4.3 Discussion of Experimental Results 64
4.5 Related Work 66
4.6 Chapter Summary 67

5 Machine Learning based Parallel SAT 68
5.1 Contribution 70
5.2 Propagation-rate 71
5.2.1 The AMPHAROS Solver 72
5.2.2 Propagation Rate Splitting Heuristic 72
5.2.3 Worker Diversification 73
5.2.4 Experimental Results 73
5.3 Machine Learning based Splitting Heuristics 76
5.3.1 The Splitting Problem 77
5.3.2 Handling Timeouts 78
5.3.3 Learning to Rank 78

5.3.4 Features for Training the Models 80
5.3.5 Training Data 82
5.3.6 Analysis of the Learned Models 82
5.4 Implementation 84
5.4.1 Implementation of Splitting in Painless-DC 84
5.4.2 Feature Computation in MapleCOMSPS for Machine Learning . . . 84
5.5 Experimental Results 85
5.5.1 Evaluation over SAT 2018 and 2019 Competition Instances 85
5.5.2 Evaluation over Cryptographic Instances 88
5.6 Related Work 90
5.7 Chapter Summary 91

6 Conclusion **92**
6.1 Overview of Results 92
6.2 Impact and Takeaways 93
6.3 Limitations 94
6.3.1 Programmatic Extension of Reasoning Components 94
6.3.2 Machine Learning based Splitting Heuristic 94
6.3.3 Heuristic Initialization using Bayesian Moment Matching 95
6.4 Future Work 95

References **97**

List of Figures

1.1 Overview of our contributions 3

2.1 Splitting tree of a formula. 13

2.2 A Beta prior is assigned to each variable in the beginning. The posteriors are then calculated each time when encountering a new clause. We project the posteriors back to Beta distributions using BMM, which serves as priors for the next clause. 19

2.3 Alternative diagram of SHA-1's round function. 22

2.4 The hardware fault injection in the input of last 16 rounds of SHA-1 and the algebraic encoding of faulty runs. 28

3.1 Block Diagram of a CDCL SAT solver with the Programmatic components that implement cryptographic related reasoning (shaded blocks). 34

3.2 A high-level diagram of the SHA-1 attack. The values δ_1 and δ_2 represents the injected faults. H denotes the correct hash output and H'_1 and H'_2 are the faulty outputs. The dashed box is the part that is being encoded into CNF. The shaded boxes are copies of the white 16 rounds, and $W_{64}, \cdots, W_{79}$ variables are shared between all of them. 42

3.3 A high-level diagram of the fault attack on SHA-256. 43

3.4 Comparison of MapleSAT and its programmatic versions on 32-bit fault model AFA on SHA-1. 46

3.5 Comparison of MapleSAT and its programmatic versions on 32-bit fault model AFA on SHA-256. 47

4.1 Overview of BMM as a component in a SAT Solver. 57

4.2 Performance comparison of MapleSAT, Glucose and CryptoMiniSAT solvers with default, and BMM initialization methods on hard cryptographic benchmarks. 61
4.3 Performance comparison of different version of MapleCOMSPS on SAT competition 2018 benchmark. 63
4.4 Comparison of total time a SAT solver took to solve an instance vs BMM preprocessing of the formula with 100 epochs on the SAT 2016 competition benchmark. 66

5.1 Performance of MapleAmpharos vs. competing parallel solvers over the SAT 2016 Application benchmark 74
5.2 Performance of MapleAmpharos vs. competing parallel solvers on SHA-1 instances 76
5.3 Percentage of instances where the predicted best variable is within the actual top-k variables for k between 1 and 10. 83
5.4 Cactus plot for performance comparison of parallel SAT solvers on filtered main track benchmark of SAT 2018. 87
5.5 Cactus plot for performance comparison of parallel SAT solvers on filtered benchmark of SAT race 2019. 88
5.6 Performance of `MaplePainless-Pairwise` against baseline Painless and `Treengeling` on cryptographic instances. 89

List of Tables

2.1 Applying column addition to multi-operand addition of five bitvectors. . . 25

3.1 The number of solved AFA instances out of 100 for different number of faults and maximal weight of the faults 45

3.2 Notation for all generalized conditions. Each character represents the set of possible values for a pair of bits. 49

3.3 CPU times (in seconds) for SAT-based differential cryptanalysis (finding collisions) in 25 rounds of SHA-256. 51

4.1 Number of solved instances out of 50 hard cryptographic instances and average runtime (in seconds) of MapleSAT, Glucose and CryptoMiniSAT with different initialization methods. 62

4.2 Number of solved instances (out of 400) and average runtime (in seconds) of MapleCOMSPS and MapleLCMDistChronoBT and their variations on SAT competition 2018 benchmark. SAT column shows how many of the solved instances were satisfiable. 64

4.3 Number of solved instances (out of 200) and average runtime (in seconds) of MapleCOMSPS and MapleLCMDistChronoBT and their variations on SAT race 2019 benchmark. SAT column shows how many of the solved instances were satisfiable. 65

5.1 Solving time details of MapleAmpharos and competing parallel solvers on SAT 2016 Application benchmark 75

5.2 Average solving time comparison on SHA-1 benchmark 77

5.3 Variable ($var_{features}(v)$) and Formula features ($formula_{features}(\phi)$). 81

5.4 Performance comparison of our solvers vs state-of-the-art divide-and-conquer parallel SAT solvers. Number of solved instances in each benchmark is out of 400, and for Total row, it is out of 800. SAT column shows the number of satisfiable instances solved (resp. UNSAT). The bold entries, show the best result on benchmark in each column. 86

Chapter 1

Introduction

Boolean satisfiability (SAT) is a fundamental problem in computer science, that asks whether there exists an assignment to variables of a Boolean formula that evaluates it to true (satisfiable). Boolean SAT solvers are programs that accept a SAT instance in conjunctive normal form and determine their satisfiability. Over the last two decades, we have seen a dramatic improvement in the efficiency of conflict-driven clause-learning (CDCL) SAT solvers, enabling them to routinely solve very large instances obtained from real-world applications. Modern SAT solvers are now well-known as powerful general purpose search tools. They have been used in solving problems from many different domains, such as verification [CGP+08], AI [Rin09], and cryptography [MZ06]. They get their power from reasoning components like clause learning [MSS99] and many different search heuristics, like VSIDS[1] [MMZ+01] or LRB[2] branching [LGPC16a], clause deletion [AS09b], restarts [AS12] and lazy data structures [MMZ+01].

The availability of such powerful search tool has led many researchers to propose the use of SAT solvers for cryptanalysis of hash functions and symmetric encryption schemes, referred to as *SAT-based Cryptanalysis* [MM00]. For example, SAT solvers are used in preimage attacks [MS13], [Nos12], collision attacks [MZ06], [Pro16], and linear and differential cryptanalysis of lightweight block ciphers[ADWL17], [KLT15]. SAT solvers are increasingly an important tool in the toolbox of the practical cryptanalyst and designer of hash functions and encryption schemes.

Although in some of the approaches, the heuristics of the solver are altered to improve their efficiency, e.g. branching heuristics [Pro16], [SZBP11] and restart policy [NLG+17],

[1] Variable State Independent Decaying Sum

[2] Learning-Rate Branching

most of these approaches used a direct encoding of the said problems into a satisfiability problem and used SAT solvers as a blackbox. The changes are limited to the search heuristics and do not alter the logic reasoning components of the solver. The one notable exception is the CryptoMiniSAT solver [SNC09], that adds reasoning over XOR clauses to the solver to improve the solving of cryptographic instances that heavily use XOR operations.

The current work on SAT-based cryptanalysis is similar to the *eager* approach in solving Satisfiability Modulo Theories (SMT) formulas, where the formula is directly translated into a SAT instance and then a SAT solver is invoked on it. The benefit of this approach is that we can use any SAT solver as-is and leverage the performance of the solver and its improvement capacity over time. The downside of this approach is the loss of the high-level semantics of the underlying theories. This means that the SAT solver needs to perform extra computations to prove facts that are readily available in the higher level logic (e.g. $x + y = y + x$ in the integer arithmetic). The other main approach to solving SMT instances, called *lazy* approach, integrates the CDCL style search with theory-specific solvers (T-solvers). This architecture is referred to as CDCL(T). Generally speaking, a T-solver is useful only if it participates in propagation and conflict analysis reasoning engines of the SAT solver they extend.

> **Thesis Statement:** Black-box SAT-based cryptanalysis has limited power. CDCL-based solvers can be enhanced with cryptographic reasoning components and tailored search heuristics in a white-box fashion for cryptanalysis tasks. These enhancements enable the CDCL SAT solvers to traverse the trade-off between the flexibility and search power of SAT solvers on one hand and the performance of dedicated tools for specific cryptanalysis tasks on the other hand.

To address this issue, we developed a set of methods that improve the state-of-the-art SAT-based cryptanalysis. We made contributions in both search heuristics and reasoning components of CDCL SAT solvers which can be divided into three main lines. First, we present a framework called CDCL(Crypto) (inspired by CDCL(T) paradigm) to extend the functionality of propagation and conflict analysis components of CDCL SAT solvers with cryptographic reasoning coming from the cryptanalysis problem being encoded into SAT. Second, we propose a machine-learning based parallel SAT solver that performs well on cryptographic problems compared to many state-of-the-art parallel SAT solvers. Finally, we used Bayesian moment matching to initialize the solver heuristics and find a promising starting point for solving a cryptographic problem.

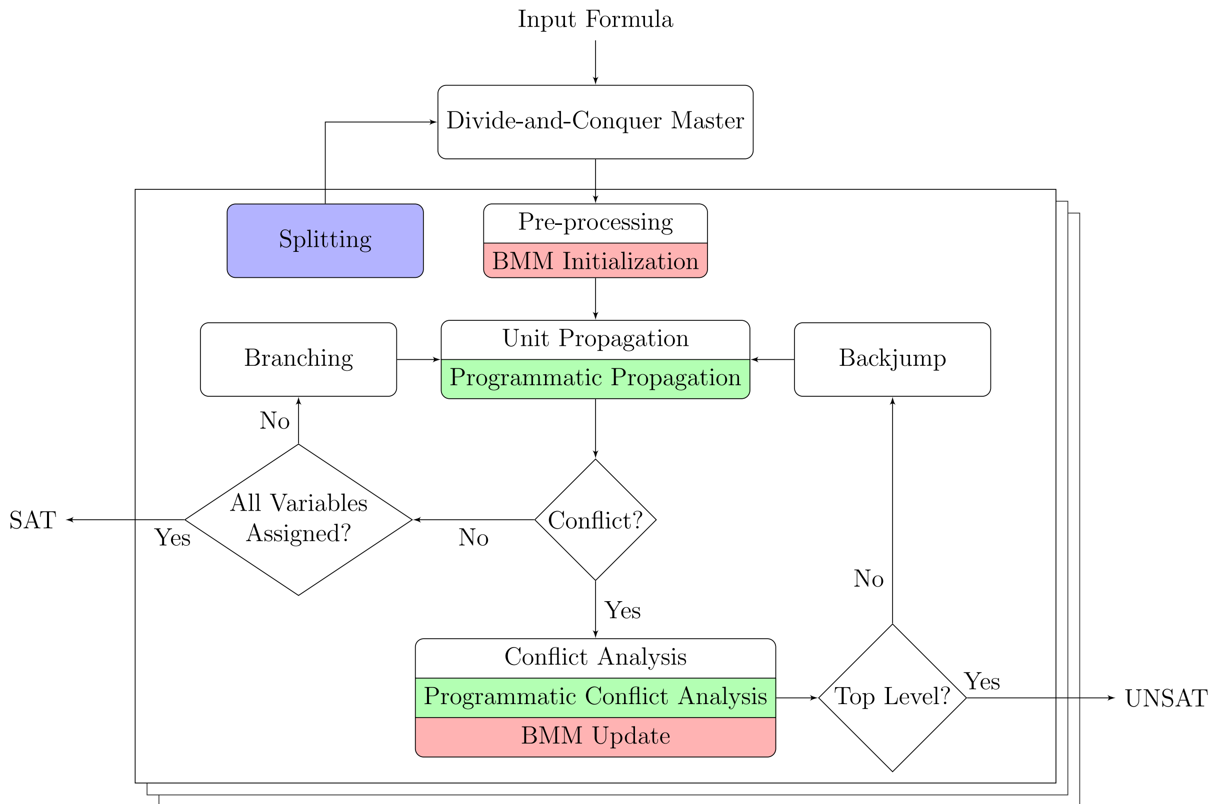


Figure 1.1: Overview of our contribution within a divide-and-conquer parallel SAT solver with sequential CDCL backend solvers. The colored blocks show the extensions to their neighboring CDCL components. [green] blocks are described in Chapter 3, [red] blocks in Chapter 4, and [blue] blocks in Chapter 5.

Figure 1.1 shows a high-level block diagram of a divide-and-conquer parallel SAT solver (a master node talking to multiple sequential CDCL backend solvers), and a peek inside a CDCL SAT solver. The figure highlights which components of the CDCL solver we have extended or improved (Pre-processing, Unit propagation, Conflict Analysis, and Splitting). The colored-dashed blocks are our extensions/implementations and the neighboring solid blocks are the original components that have been enhanced. The contributions are color-coded with the respective chapter that describes them.

1.1 Extending Reasoning Components for Cryptographic Problems

Although modern CDCL solvers are capable of handling a large number of constraints coming from many different domains, there are still many problems that could benefit from the search capabilities of SAT solvers, but face a representational challenge. In converting these problems into Boolean logic, either some high-level properties are lost, or a blowup in problem size happens when the desired properties are also encoded. Some of the basic operations in cryptographic primitives fall into this category of problems. In other words, there are some implications in the original problem that are not found by the implication engine in the Boolean level. This problem especially affects instances encoding symmetric block cipher and cryptographic hash functions, because they are constructed by repeating a small round function several times to achieve high levels of diffusion. Thus a lost implication at a round function level can break a long chain of implications.

In this work, we present a method and a prototype tool, called CDCL(Crypto), that uses a programmatic SAT design to extend the functionality of propagation and conflict analysis components with the domain knowledge of the cryptographic problems. We demonstrate the power of this framework in two cryptanalysis tasks. First, we show how we can speed up algebraic fault attack on SHA cryptographic hash functions, and effectively reducing the number of required faults to recover the embedded secret message bits in a hardware implementation of the said functions. Second, we improve the results on state-of-the-art SAT-based differential cryptanalysis of SHA-256. In each of these tasks, the propagation and conflict analysis are programmatically extended specific to that problem. However, the underlying framework is exactly the same. This shows that cryptographers can use this tool to program their cryptanalysis techniques on top of a flexible and powerful search engine.

1.2 Improving Splitting Heuristics in Parallel SAT Solvers

There has been a great body of work on sequential Boolean SAT solvers. The emergence of many-core machines, however, opens new possibilities in this domain, and parallel SAT algorithms constitute a natural next step in SAT solver research. Unfortunately, developing practically efficient parallel SAT solvers that scale well with an increasing number of cores has shown to be a much harder challenge than anticipated. Furthermore, there are very

few prior works on parallel SAT solvers that target cryptographic instances. The two most widely used architectures for parallel SAT solvers are portfolio and divide-and-conquer (and its variants) approaches.

For our second line of contribution, we focused on divide-and-conquer solvers. These types of solvers split the formula into smaller sub-formulas and solve the resultant sub-formulas in parallel using sequential CDCL solvers. We studied the splitting heuristic in divide-and-conquer solvers. Splitting heuristics pick a variable and set it to both True and False within the original formula, thus generating two sub-formulas. Effectively they try both branches of the search space branched at the splitting variable in parallel. This process can be further applied to the generated sub-formulas to create many sub-formulas. The goal of splitting heuristics is to identify the variables that generate easier sub-formulas for CDCL backend solvers, in other words, they try to answer this question: *How to divide, so the conquer becomes easier?*

In this work, we first present an ad-hoc splitting heuristics based on how well setting the splitting variable propagates to other variables and generates smaller sub-formulas. We show that this heuristic performs very well on cryptographic instances compared to other state-of-the-art parallel SAT solvers. Next, we look at the problem more abstractly and frame the splitting problem as a ranking problem. We give a quality metric for splitting candidates to define the splitting as a runtime optimization problem more formally. We then present a machine learning technique that generates a ranking of variables according to their splitting quality. We show that in an apple-to-apple comparison with other splitting heuristics, we improve the performance of the baseline solver over application benchmarks of the SAT competition, and solve more instances than leading divide-and-conquer solvers in the competition.

1.3 Initializing SAT Solver Heuristics

It is well-known that the *initial order and value assignment to the variables of an input formula* can have a huge impact on the performance of a CDCL SAT solver. By initial order, we mean the order chosen by the solver at the beginning of its search before making the first decision (we can similarly define the notion of initial value assignment). The problem then is "find the optimal order (resp., value assignment) for a given input formula such that the solver's runtime is minimized".

In this work, we used the encoding of SAT problem into a Bayesian inference setting. We start from a random prior describing an assignment to the variables of the given

formula. Each formula clause is then observed as evidence, and the distribution is updated using Bayesian inference and moment matching. Bayesian inference by itself could result in a mixture model that grows exponentially in the number of models. We use moment matching to approximate a distribution with a single model. Bayesian moment matching finds a posterior distribution that approximates a solution to the input SAT formula, that ideally satisfies most of the clauses (if not all of them). This method might not scale to solve large SAT instances, however, we use the posterior to find a close-to-solution starting point. We pre-process the formula and encode the derived posterior probabilities into initial values for value selection (polarity heuristic) and variable order (branching heuristic). We further improve the initialization by using newly generated clauses (implied by the formula) during the search and guide the search toward paths that are highly likely leading to a solution. This formulation is very well-suited for satisfiable instances because the posterior is describing a solution. All of our cryptographic problems are satisfiable (there exists a secret key or hash preimage, and the task is to find the solution), therefore we studied this technique on cryptographic instances as an important class of satisfiable instances.

We show that with the combination of polarity and branching initialization, not only we get a significant speedup in cryptographic instances, but also we get considerable performance improvements in other application instances, compared to 4 other initialization methods that we have experimented with.

1.4 Supporting Code Contributions

This dissertation is partially based on the following codes:

- CDCL(Crypto): We developed a programmatic SAT system called CDCL(Crypto), which provides programmatic propagation and conflict analysis callbacks on top of MapleSAT SAT solver [LGPC16a]. The system callbacks contain pre-implemented propagation for multi-operand addition and SHA message recovery, and can easily be adapted to other use cases.
Source code: https://github.com/saeednj/CDCL-Crypto.
- *MapleAmpharos* and *MaplePainless*: We developed a parallel SAT solver called MapleAmpharos, that implements propagation-rate heuristic on top of Ampharos parallel SAT solver [ALST16]. Also developed two machine learning based ranking methods for splitting heuristic on top of Painless parallel SAT framework [LFBSK19].
Source code: https://github.com/saeednj/MaplePainless-DC.
- *BMM-SAT*: We instrumented MapleSAT, Glucose, CryptoMiniSAT, MapleCOMSPS and MapleLCMDistChronoBT to add Bayesian moment matching based polarity and variable order initialization.
Source code: https://github.com/saeednj/BMMSAT.
- *SAT Encoding*: We developed an encoder for translating various cryptanalysis tasks into SAT instances.
Source code: https://github.com/saeednj/SAT-encoding.

1.5 Supporting Publications

This dissertation contains material from the following published, under review documents:

[NNS+17] S Nejati, Z Newsham, J Scott, JH Liang, C Gebotys, P Poupart, V Ganesh.
A propagation rate based splitting heuristic for divide-and-conquer solvers.
International Conference on Theory and Applications of Satisfiability Testing (SAT) 2017.

[NLG+17] S Nejati, JH Liang, C Gebotys, K Czarnecki, V Ganesh.
Adaptive Restart and CEGAR-based Solver for Inverting Cryptographic Hash Functions.

Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE) 2017.

[NHGG18] S Nejati, J Horáček, C Gebotys, V Ganesh.
Algebraic Fault Attack on SHA Hash Functions Using Programmatic SAT Solvers.
International Conference on Principles and Practice of Constraint Programming (CP) 2018.

[NG19] S Nejati, V Ganesh.
CDCL(Crypto) SAT Solvers for Cryptanalysis.
Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (CASCON) 2019.

[NDT+20] S Nejati/H Duan, G Trimponias, P Poupart, V Ganesh.
Online Bayesian Moment Matching based SAT Solver Heuristics.
Submitted to ICML 2020.

[NLFG20] S Nejati, L Le Frioux, V Ganesh.
A Machine Learning based Splitting Heuristic for Divide-and-Conquer Solvers.
In preparation for CP 2020.

Chapter 2

Background

In this chapter we introduce definitions and preliminary material for the following chapters. The three main lines of preliminary materials that is covered in this chapter are from Parallel SAT solving (Sections 2.1,2.2,2.4), Machine Learning (2.5.1,2.5.2), and Cryptography (Sections 2.6,2.8,2.9) topics that are related to our work.

2.1 Boolean SATisfiability

A Boolean variable is a variable that only accepts True and False values. A *literal* (l) is a Boolean variable (v) or its negation ($\neg v$). A *clause* (C) is a disjunction of literals ($l_1 \vee l_2 \vee \cdots \vee l_n$). A clause with a single literal is called a *unit clause*. A Boolean formula (ϕ) in *conjunctive normal form* (CNF) is a conjunction of clauses ($C_1 \wedge C_2 \wedge \cdots \wedge C_m$). *Boolean satisfiability problem* is, given a Boolean formula determine if it is satisfiable, in other words: is there an assignment to the Boolean variables in the formula that makes the formula evaluate to True? A Boolean SAT solver is a program that determines the satisfiability of a given Boolean formula, typically in CNF.

We refer to set of all of the variables (resp. clauses) in a formula ϕ, using $vars(\phi)$ (resp. $clauses(\phi)$). For a given formula ϕ, an *assignment* α, to the variables of ϕ, is a mapping $\alpha : vars(\phi) \rightarrow \{False, True\}$. An assignment is *complete* when all of variables are assigned a truth value, otherwise it is *partial*. A literal l (resp. $\neg l$) is satisfied by α, if it maps it to true (resp. false). A clause is satisfied by *alpha*, if at least one of the literals in that clause is satisfied by α. We denote *simplification* of formula ϕ over assignment α with $\phi[\alpha]$, which means removing all satisfied clauses by α and all falsified literals by α in the remaining clauses of ϕ.

Algorithm 1 Pseudocode of a basic CDCL SAT solver.

```
Input: CNF formula φ
Output: SAT/UNSAT
 1: function CDCL(φ)
 2:     T ← {}                                  ▷ trail of decision and implied literals.
 3:     while true do
 4:         T ← unitPropagate(φ, T)
 5:         if (φ, T) is in conflict then        ▷ unit propagation derives an empty clause.
 6:             if T = {} then                   ▷ the conflict is at the top decision level.
 7:                 return UNSAT
 8:             c ← analyzeConflict(φ, T)        ▷ derive a conflict clause.
 9:             m ← assertionLevel(c)
10:             T ← T_m       ▷ remove any decisions/propagations after decision level m.
11:             φ ← φ ∪ {c}
12:         else
13:             if RestartCondition() = true then
14:                 T ← {}
15:             l ← pickBranchingLiteral()
16:             if l = null then                 ▷ there is no unassigned variables.
17:                 return SAT
18:             T ← T ∪ {l}                      ▷ add the decision literal to the trail
```

2.2 Conflict-Driven Clause-Learning SAT Solvers

We refer the readers to [BHvM09] for a detailed description of conflict-driven clause-learning (CDCL) SAT solvers and only review a high-level overview of these solvers in this section. CDCL solvers traverse the search space of assignments to the variables of the given formula by setting values to variables (decision), finding implied literals (unit propagation), pruning the search space by learning from conflicting decisions (clause learning) and backtracking to undo the bad decisions. The pseudocode of a basic CDCL SAT solver is given in Algorithm 1.

Branching. The solver picks an unassigned variable (*variable selection heuristic*) and assign a value to it (*polarity heuristic*). Typically the combination of these two is referred to as *branching heuristic*. The heuristically picked variable is called a *decision variable*, and together with its value, it is referred to as *decision literal*. The decision literal is then appended to a sequence called *trail*. The *decision level* of a literal on the trail is defined as the number of decision variables that come before it on the trail.

Unit Propagation. A unit clause is satisfied when the literal in it is satisfied. During the search, if all but one of the literals in a clause are set to false, the clause becomes unit and implies that the unassigned literal should be set to true. These last literals are referred to as *implied literals*. The implied literals are added to the trail with the decision level set to the same value as the last decision level in the trail. The solver simplifies the formula after setting literals in unit clauses to true, by removing the satisfied clauses and falsified literals. This simplification might create new unit clauses, where the simplification can be applied again. This process is repeated until there are no implied literals.

Conflict Analysis. If during the unit propagation, all of the literals in a clause are falsified, that clause is falsified with respect to the current partial assignment. In other words, if the simplification removes all of the falsified literals and derives a clause that is empty, we are in a conflicting state. If the trail is empty, then the formula is unsatisfiable by itself. Otherwise a *conflict analysis* routine, looks at the decisions and implications that led to this conflict and derives a *conflict clause* that explains the cause of this conflict. The process of deriving this particular clause is referred to as *clause learning*, and thus the derived clause is also called a *learnt clause*. The assertion level m is defined as the second highest decision level of the literals appearing in the conflict clause. The solver *backjumps* to the decision level m, by removing any literals in the trail that has a decision level greater than m. The learnt clause is implied by the formula, therefore it can be added to the formula. This learnt clause prevents the solver from getting to the same partial assignment in the future.

Restart. A search restart is clearing the trail entirely while retaining the learnt clauses. When the unit propagation terminates without a conflict, the solver may choose to restart. Technically, a restart is a backjump to decision level 0. The choice of whether to perform a restart or not is typically guided by a heuristic.

2.3 Arc Consistency and SAT

The following definition is adapted from general arc consistency definition given in [BBR09].

Definition 2.3.1. Let R be an inference rule of propositional logic. Let ϕ be a Boolean formula which encodes a constraint C in CNF. We say that the encoding of C into ϕ R-**maintains Generalized Arc Consistency (GAC)** if for all partial assignments α, i.e.,

a conjunction of literals, and for all literal ℓ the following holds

$$C \wedge \alpha \vdash \ell \Rightarrow \phi \wedge \alpha \vdash_R \ell \quad \text{(i.e., } \ell \text{ is derived from } \phi \wedge \alpha \text{ by } R\text{).}$$

The following example illustrates the fact that some encodings do not maintain GAC under unit propagation (UP), which is the propagation procedure in CDCL SAT solvers.

Example 2.3.1. Consider the pseudo-Boolean constraint: $x + y \leq 0$, i.e., $x, y \in \{0, 1\}$ and "+" denotes integer addition. We can encode this constraint into a CNF formula ϕ by using a half-adder with inputs x and y and forcing the outputs to be zero. The half-adder relations for carry and sum outputs c and s are $x \wedge y$, $x \oplus y$. The final encoding of $C = (c \leftrightarrow x \wedge y) \wedge (s \leftrightarrow x \oplus y) \wedge (\neg s \wedge \neg c)$ in CNF is $\phi = (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$. It is clear that x and y should be set to zero. But these values are not discovered by applying UP on ϕ.

One would naturally expect that the assignment α to the input variables, fully unit propagate to the output bits, but this may not always be the case and depends on the encoding ϕ.

2.4 Parallel SAT Solvers

The general approach for utilizing many-core systems for CDCL SAT solvers is to have several sequential CDCL solvers as *workers* running on different computing cores and have them work cooperatively or competitively (or a mix of both) to solve the input problem. There are two main strategies that parallel SAT solvers use: *portfolio* and *divide-and-conquer*.

Portfolio. In parallel portfolio solvers, the original formula is given to all of the workers. Portfolio solvers rely on two main techniques, namely: *diversification*, where the worker solvers are configured with different heuristics and/or initial parameters and *clause sharing*, where workers periodically share the learnt clauses they generate during the search. Diversification helps the solvers to dive into different parts of the search space and make use of the complementary power of different heuristics. Clause sharing helps to have useful information about each sub-space in all solvers, thus alleviating the need to explore any sub-space multiple times. The input formula is SAT (resp. UNSAT) if any of the solvers return SAT (resp. UNSAT).

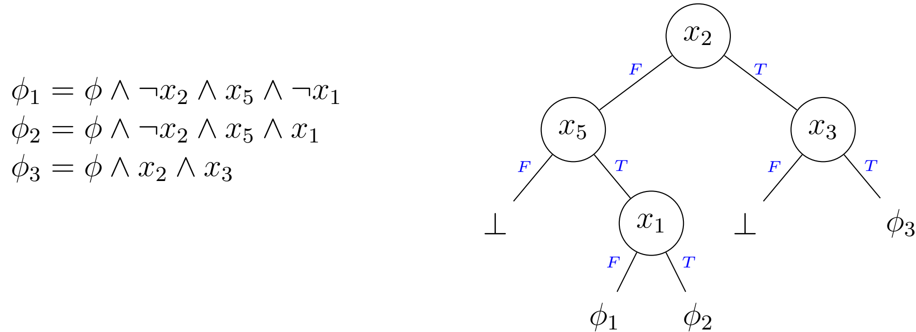


Figure 2.1: Splitting a formula ϕ into smaller sub-formulas ϕ_1, ϕ_2 and ϕ_3, and the corresponding splitting tree. Each node is a variable in ϕ and edges labeled with F are setting that variable to False (and T for True). Each sub-formula is a the original formula constrained with a cube (conjunction of literals from root to the corresponding leaf). Some paths could be unsatisfiable.

2.4.1 Divide-and-Conquer Solvers

Divide-and-conquer solvers first split the search space of the input formula into many smaller sub-spaces and then solve the resulting sub-formulas in parallel using CDCL solvers. The input formula is SAT if one of the worker solvers returns SAT. However, to prove that the input formula is UNSAT, all of the workers need to return UNSAT for their sub-formulas.

Search Space Splitting

There are different approaches to search space splitting. Commonly the splitting is done by a function which maps a formula ϕ to n constraints $p_1, \cdots, p_n$, such that, $\phi \equiv (\phi \wedge p_1) \vee \cdots \vee (\phi \wedge p_n)$ (sub-formulas cover the search space of the input formula), and $\phi \wedge p_i \wedge p_j$ is unsatisfiable for all $i \neq j$. The process of splitting the formula can be done statically (all of the constraints are generated together), or iteratively (the formula is split into two sub-formulas and the splitting is recursively applied to the sub-formulas) [HM12]. A common splitting strategy is *guiding path* [ZBH96]. The splitting is done iteratively and at each splitting point, the formula ϕ is divided into two sub-formulas $\phi_1 = \phi \wedge x$ and $\phi_2 = \phi \wedge \neg x$, where $x \in vars(\phi)$. We refer to x as the *splitting variable*. Usually, these solvers are implemented in a master-slave architecture, where the master maintains the search space splitting in the form of a binary tree (*splitting tree*). Each node in this tree is a splitting variable. The left branch represents setting parent variable to False and right branch is for

setting it to True. The path from the root to each of the leaves represents a conjunction of literals called *cube* (representing a splitting constraint p_i). Figure 2.1 shows a splitting tree and its corresponding cubes. Each sequential solver receives the formula with literals in its cube as an assumption set.

Splitting Heuristic

The splitting heuristic is the problem of choosing a splitting variable. The target of splitting heuristic is to generate balanced sub-spaces and reduce the overall runtime of the solving process. Splitting heuristics are commonly one of these two main approaches: *Look-ahead* and *Look-back*.

Look-ahead. This type of heuristics looks into the question of "what would happen if we pick variable x for splitting?" and picks the variable that has the largest effect. The difference between original formula and formula after simplifying it over both x and $\neg x$ (picking x for splitting) is measured using a so-called *DIFF metric*, which for example computes a weighted average of reduced clauses (clauses that were touched and not satisfied). Another heuristic (*MIXDIFF metric*) mixes the values for x and $\neg x$ branches to arrive at a single value for each variable x. Analyzing all of the variables to compute their value can become expensive as the number of variables increases. *Cube-and-Conquer* solvers [HKWB11] use look-ahead heuristics for splitting the formula into many smaller sub-formulas and solve them using CDCL solvers in parallel. The main benefit of performing look-ahead first and CDCL next is that look-ahead techniques scan a big subset (if not all) of the variables, and this provides them with a global view of the formula, compared to CDCL solvers that analyze the formula very locally. In other words, using a global view they split the formulas into very focused and compact sub-spaces that can be handled very easily by CDCL solvers.

Look-back. This approach looks at how important is a variable when participated in the search exploration (according to some heuristic measure, e.g. contribution to clause learning). We let the sequential solvers work on their sub-formulas and gather search related statistics. If the sub-formula is easy, the solver returns a result, otherwise, we need to break it down, and at this point, the solver looks back at the gathered statistics and according to the splitting heuristics picks the best splitting variable. The main benefit of these types of heuristics is their dynamic nature that adapts to the problem and the statistics are often very cheap to compute and handle. For example, Ampharos [ALST16]

uses VSIDS activities (maintained for branching). Painless-DC [LFBSK19] uses number of *flips* for each variable. A flip is the number of times that unit propagation sets a variable to a value that is different than its previous propagated value.

The notation $t_S(\phi)$ refers to the time to solve a Boolean formula ϕ with a sequential worker CDCL SAT solver S (We drop the subscript if it is clear from context). We denote the reduced formula after setting v to False (respectively to True) with $\phi[v = F]$ (respectively, $\phi[v = T]$). By reducing a formula we mean simplification via unit propagation (i.e., removal of satisfied clauses from the formula, falsified literals from clauses).

Performance Metric. The term *performance metric*, with respect to a given solver S, refers to a function $pm : \phi \times v \to \mathbb{R}$, over a formula ϕ and a variable $v \in vars(\phi)$, that characterizes the "quality" of splitting ϕ over v. Minimizing this metric ideally should correlate with minimizing solver runtime.

More precisely, the general goal of designing a splitting heuristic is twofold: first, to come up with a metric that correlates with minimizing solver runtime, and second to design a function to compute said metric. Researchers have proposed a variety of performance metrics in the context of splitting heuristics. Below are definitions of three such performance metrics and the intuition behind each of them. In previous work, researchers have found that these metrics are good proxies for minimizing runtime in the context of splitting in DC solvers. Further, to state the obvious, it is ideal to split on a variable that minimizes these metrics over all variables of an input formula. Let $\phi_1 = \phi[v = F]$ and $\phi_2 = \phi[v = T]$, be the sub-formulas after splitting ϕ over v.

- $pm_1(\phi, v) = \max\{t(\phi_1), t(\phi_2)\}$: This metric aims to capture the runtime of a DC solver executed in parallel over the sub-formulas ϕ_1 and ϕ_2.
- $pm_2(\phi, v) = t(\phi_1) + t(\phi_2)$: This function gives higher priority to splitting variables that make the problem easier even in a single core setting.
- $pm_3(\phi, v) = -(t(\phi) - t(\phi_1)) \cdot (t(\phi) - t(\phi_2))$: The idea behind this metric is to measure runtime "progress" in each branch (by comparing the runtime of sub-formulas with the original formula) and also aims to balance the hardness of the two branches.

2.5 Machine Learning

2.5.1 Supervised Learning

Consider a function $f : D \to R$. Supervised learning is a machine learning method of learning a function that maps inputs from domain D ($x \in D$) to the outputs from range R ($y = f(x) \in R$), based on given input-output examples. Each example pair $< x, f(x) >$ used for learning is called a training example, and a set of these examples is referred to as *labeled training data*. In other words, supervised learning is the process of inferring a function $\tilde{f}$ from labeled training data, which approximates the function f. We say $\tilde{f}$ *fits* the training data if the learned function outputs the correct values for input values from training data with a high probability. The hope is that $\tilde{f}$ is not only fitting the training data but also can correctly find the output of unseen examples or in other words *generalizes*.

Supervised learning can be categorized into two types of *Regression* and *Classification*, based on the range of the output. In regression, the output is a continuous value, and in classification, we have categorical output. In this work, we only work with classification algorithms and more specifically binary classification, where the function that we want to learn has a signature of $f : D \to \{0, 1\}$. To learn a binary classifier we used *logistic regression* and *random forest* techniques.

Logistic regression [Cox58] is of type $\tilde{f} : \mathbb{R}^n \to [0, 1]$, where the input is vector of features extracted from the example input object and the output is the probability of class 1. The function implements a linear regression (a weighted sum of input values), followed by a sigmoid function σ to squeeze the output to be between 0 and 1.

$$\tilde{f}([x_1, x_2, \cdots, x_n]) = \sigma(w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n), \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

Random forest [LW+02] is an ensemble learning method, that constructs a set of decision trees at training time and outputs the class that appears most often as the output of decision trees. Decision trees are a popular method for various machine learning tasks. However, trees that are grown very deep tend to learn highly irregular patterns: they overfit their training sets, i.e. have a low bias, but very high variance. Random forests are a way of averaging multiple deep decision trees, trained on different parts of the same training set, with the goal of reducing the variance. This comes at the expense of a small increase in the bias and some loss of interpretability, but generally greatly boosts the performance in the final model.

2.5.2 Bayesian Moment Matching and SAT

In this section, we give an overview of the Bayesian moment matching algorithm using the naïve Bayes model.

Bayesian Inference. Bayesian inference is a method of statistical inference, where the Bayes' theorem is used to update the probability distribution of a hypothesis as more and more evidence about that hypothesis becomes available. Bayesian inference derives a posterior distribution $P(H|E)$ about a hypothesis H, given a prior distribution $P(H)$ and a likelihood function $P(E|H)$, after observing an evidence E.

$$P(H|E) = \frac{P(E|H).P(H)}{P(E)}$$

Method of Moments. The method of moments is a statistical method to estimate the parameters of a population. Moments are the expected values of the powers of the random variables under consideration $(E[X], E[X^2], \cdots)$. This method first expresses the moments as a function of parameters of interest. Then each expression is set equal to (matched with) the sample moments. Solving this equation set gives us an estimation of the parameters. The number of equations in this equation system is equal to the number of parameters that we want to estimate.

Bayesian Moment Matching. Bayesian Moment Matching (BMM) for mixture models was proposed to prevent the exponential growth of mixture components in online Bayesian learning [JP16, RZP16]. A distribution belonging to the same family as the prior is used to approximate the posterior by matching the sufficient moments, in order to reduce the complexity of posterior distributions. BMM has been successful in the context of topic modelling [Oma16, HP16], hidden Markov models [JCC+16] and sum-product networks [RZP16].

Problem Setup. Let Z represent a binary hidden variable and X represent a binary observable variable. Let $\{X_1, \ldots, X_n\}$ be a set of binary $i.i.d$ observations from X. The conditional distribution of $X|Z$ is completely known. We use c_1 to denote $P(X = 0|Z = 0)$ and c_2 to denote $P(X = 0|Z = 1)$. Let θ represent the unknown probability of the hidden variable $P(Z = 0)$, the quantity we wish to infer from $\{X_1, \ldots, X_n\}$ in an online and Bayesian fashion.

Let $P_k(\theta)$ be the probability of θ after observing k evidences and consider a beta distribution as the prior over θ. More specifically, $P_0(\theta) = \frac{1}{B(\alpha_0,\beta_0)}\theta^{\alpha_0-1}(1-\theta)^{\beta_0-1}$, where $B(\alpha_0, \beta_0)$ represents a beta function of α_0 and β_0. The posterior after observing the first evidence is:

$$\begin{aligned} P_1(\theta|X_1 = 0) &\propto P_0(\theta)P(X_1 = 0|\theta) \\ &\propto \theta^{\alpha_0-1}(1-\theta)^{\beta_0-1}[\theta c_1 + (1-\theta)c_2] \\ &\propto c_1\theta^{\alpha_0}(1-\theta)^{\beta_0-1} + c_2\theta^{\alpha_0-1}(1-\theta)^{\beta_0} \end{aligned}$$

$$P_1(\theta|X_1 = 1) \propto (1-c_1)\theta^{\alpha_0}(1-\theta)^{\beta_0-1} + (1-c_2)\theta^{\alpha_0-1}(1-\theta)^{\beta_0} \tag{2.1}$$

The equation 2.1 shows that the posterior is a mixture of two beta distributions after the first point is observed. Therefore, the number of mixture components in the posterior distributions will grow exponentially by a factor of two for each new observation, which makes inference intractable. To solve this problem, BMM approximates the mixture posterior $P_1(\theta)$ with a single Beta distribution $\tilde{P_1}(\theta) = Beta(\tilde{a_1}, \tilde{b_1})$ by matching the first and second moments. $\tilde{a_1}$ and $\tilde{b_1}$ can be derived by solving the equations system below:

$$\begin{aligned} \frac{\tilde{a_1}}{\tilde{a_1} + \tilde{b_1}} &= \mathbb{E}_{\theta|X_1}(\theta) \\ \frac{\tilde{a_1}(\tilde{a_1}+1)}{(\tilde{a_1} + \tilde{b_1})(\tilde{a_1} + \tilde{b_1} + 1)} &= \mathbb{E}_{\theta|X_1}(\theta^2) \end{aligned}$$

where, $\mathbb{E}_{\theta|X_1}(f(\theta)) = \int f(\theta)P_1(\theta|X_1)d\theta$

Bayesian formulation for SAT. Poupart, Jaini and Duan introduced a novel Bayesian perspective to solve the SAT problem [NDT+20]. In their Bayesian formulation, each variable in the SAT formula is a Bernoulli random variable with an unknown probability being assigned to T (true) and each clause is treated as evidence. The objective is to learn the unknown probability associated with each variable by BMM, which is illustrated with a toy SAT instance:

$$\begin{aligned} &C_1 : x \vee y \vee \neg z \\ &C_2 : x \vee y \vee z \\ &C_3 : x \vee \neg y \vee z \\ &C_4 : \neg x \vee \neg y \vee \neg z \end{aligned}$$

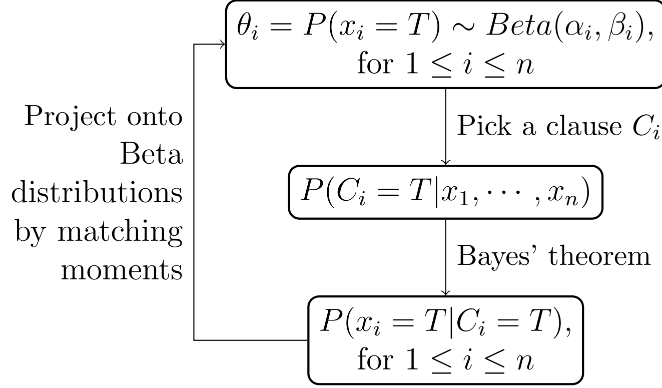


Figure 2.2: A Beta prior is assigned to each variable in the beginning. The posteriors are then calculated each time when encountering a new clause. We project the posteriors back to Beta distributions using BMM, which serves as priors for the next clause.

They use $\theta_x, \theta_y, \theta_z$ to denote $P(x = T), P(y = T), P(z = T)$ respectively. To estimate $\theta_x, \theta_y, \theta_z$ by Bayesian inference, they assume that each of them is initially distributed according to a Beta prior and that they are mutually independent. Concretely, the prior for the joint distribution is:

$$P(\theta_x, \theta_y, \theta_z) = \prod_{i=x,y,z} Beta(\theta_i; \alpha_i, \beta_i).$$

An instance is satisfiable if all of its clauses are satisfied. To satisfy a clause at least one of the literals needs to be satisfied, which can be done in many different ways if we have many literals in a clause. However, there is only one way to falsify the clause. Therefore, they define the likelihood function as the complement probability of falsifying the observed clause. For example, to falsify clause C_1, we should have $x = F, y = F, z = T$, and thus: $P(C_1|\theta_x, \theta_y, \theta_z) = 1 - (1-\theta_x) \cdot (1-\theta_y) \cdot \theta_z$. The posterior after seeing the first clause C_1 is:

$$\begin{aligned}
&P(\theta_x, \theta_y, \theta_z|C_1) \propto P(\theta_x, \theta_y, \theta_z)P(C_1|\theta_x, \theta_y, \theta_z) \\
&\propto P(\theta_x, \theta_y, \theta_z)[1 - (1-\theta_x)(1-\theta_y)\theta_z] \\
&\propto Beta(\theta_x; \alpha_x, \beta_x) \cdot Beta(\theta_y; \alpha_y, \beta_y) \cdot Beta(\theta_z; \alpha_z, \beta_z) \\
&- \frac{\beta_x}{\alpha_x + \beta_x} \frac{\beta_y}{\alpha_y + \beta_y} \frac{\beta_z}{\alpha_z + \beta_z} Beta(\theta_x; \alpha_x, \beta_x + 1) \\
&\cdot Beta(\theta_y; \alpha_y, \beta_y + 1) \cdot Beta(\theta_z; \alpha_z + 1, \beta_z)
\end{aligned}$$

Since the likelihood $1 - (1-\theta_x)(1-\theta_y)\theta_z$ can also be expressed as the sum of joint probabilities, we can see that the posterior is a mixture of products of Beta distributions. The

number of mixture components will grow exponentially as more clauses are encountered. To solve this intractability issue, we approximate the true mixture $P(\theta_x, \theta_y, \theta_z|C_1)$ by a single product of Beta distributions using BMM:

$$\tilde{P}(\tilde{\theta}_x, \tilde{\theta}_y, \tilde{\theta}_z) = \prod_{i=x,y,z} Beta(\tilde{\theta}_i; \tilde{\alpha}_i, \tilde{\beta}_i)$$

The parameters $\tilde{\alpha}_x, \tilde{\beta}_x$ are then computed by matching the first and second moments of the marginal distribution of θ_x:

$$\mathbb{E}_{\tilde{\theta}_x \sim Beta(\tilde{\theta}_x; \tilde{\alpha}_x, \tilde{\beta}_x)}[\tilde{\theta}_x] = \mathbb{E}_{\theta_x \sim P_{\theta_x}(\theta_x|C_1)}[\theta_x]$$
$$\mathbb{E}_{\tilde{\theta}_x \sim Beta(\tilde{\theta}_x; \tilde{\alpha}_x, \tilde{\beta}_x)}[\tilde{\theta}_x^2] = \mathbb{E}_{\theta_x \sim P_{\theta_x}(\theta_x|C_1)}[\theta_x^2],$$

where $P_{\theta_x}(\theta_x|C) = \int_0^1 \int_0^1 P(\theta_x, \theta_y, \theta_z|C) d\theta_y d\theta_z$. The parameters $\tilde{\alpha}_y, \tilde{\beta}_y, \tilde{\alpha}_z, \tilde{\beta}_z$ are computed similarly. Subsequently, $\tilde{P}(\tilde{\theta_x}, \tilde{\theta_y}, \tilde{\theta_z})$ is used as the prior when C_2 is observed. During one epoch, the above update is repeated once for each clause.

2.6 Cryptographic Hash Functions

A hash function maps an arbitrary length input string to a fixed length output string. There are three main properties that are desired for a cryptographic hash function [LM93]. Informally, they are:

- *Preimage Resistance*: Given a hash value H, it should be computationally infeasible to find a message M, where $H = hash(M)$.
- *Second Preimage Resistance*: Given a message M_1, it should be computationally infeasible to find another message M_2, where $hash(M_1) = hash(M_2)$ and $M_1 \neq M_2$.
- *Collision Resistance*: It should be computationally infeasible to find a pair of messages M_1 and M_2, where $hash(M_1) = hash(M_2)$ and $M_1 \neq M_2$. (There is a subtle difference between second pre-image resistance and collision resistance, in that the message M_1 is not fixed in the case of collision resistance).

Preimage resistance implies that the hash function should be hard to invert. The terms preimage attack and inversion attack are used interchangeably. Usually standard cryptographic hash functions at their core have a compression function, which takes as

input a fixed length input and outputs a fixed length (with smaller length) output. For making a hash function able to accept arbitrary long messages as input, one can use *Merkle-Damgard* (MD) construction, where the compression functions are chained together, each processing a block of the input. It is shown that if one block is collision resistant, then the whole structure is collision resistant [Mer89]. More formally a t-bit compression function is an efficiently computable function $F : \mathbb{F}_2^t \times \mathbb{F}_2^b \to \mathbb{F}_2^t$, that maps a t-bit chaining value H_{i-1} and a b-bit message block M_i to a t-bit output chaining value H_i ($H_i = F(H_{i-1}, M_i)$). MD construction breaks the input message M into fixed equal sized blocks M_i and repeatedly applies the compression function F. The initial chaining value (H_0) is a fixed number, known as *initialization vector* (IV), which is defined in the description of the hash function. For a t-bit compression function, the generic complexity of finding a collision is $2^{t/2}$ and 2^t for preimage. In the context of MD-based hash functions (e.g. SHA-1, SHA-2), the collision attack can be relaxed in terms of constraints on the input chaining value. The task is to find M_1 and M_2, $M_1 \neq M_2$ such that $hash(CV_1, M_1) = hash(CV_2, M_2)$. For collision we should have: $CV_1 = CV_2 = IV$, for *Semi-Free-Start collision* we should have: $CV_1 = CV_2$, and for *Free-Start collision* there is no constraint on CV_1 and CV_2.

2.6.1 Description of SHA-1

SHA-1 was designed by NSA and standardized by NIST in 1995 (see the standard in [FIP11]). It was widely used in many applications, but after the recent full collision reported in [SBK+17], security practitioners moved away to stronger alternatives such as SHA-2 or SHA-3, although SHA-1 seems to be still resistant against preimage and second preimage attacks.

SHA-1 uses the Merkle-Damgård construction, where each block has 512 bits. Each block is given to a compression function that outputs 160 bits, which is used as part of the input to the next block. We recall only a part of the SHA-1 specification. For the full description of SHA-1, we refer to [FIP11]. The internal state of SHA-1 is 160 bits. More precisely, five 32-bit words $a_i, \ldots, e_i$ for each round i. There are 80 rounds, and in each round a 32-bit message word W_i will be mixed in to update the state bits. The *round function* for the round $i = 0, \ldots, 79$ is defined as follows

$$(a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1}, e_{i+1}) \leftarrow \big(f(b_i, c_i, d_i) \boxplus e_i \boxplus (a_i \lll 5) \boxplus W_i \boxplus K_i, a_i, b_i \lll 30, c_i, d_i\big), \quad (2.2)$$

where $\lll$ is left rotation, $\boxplus$ is addition modulo 2^{32} and K_i is the round constant. The function f is a Boolean map operating on three 32-bit words and generating a 32-bit word.

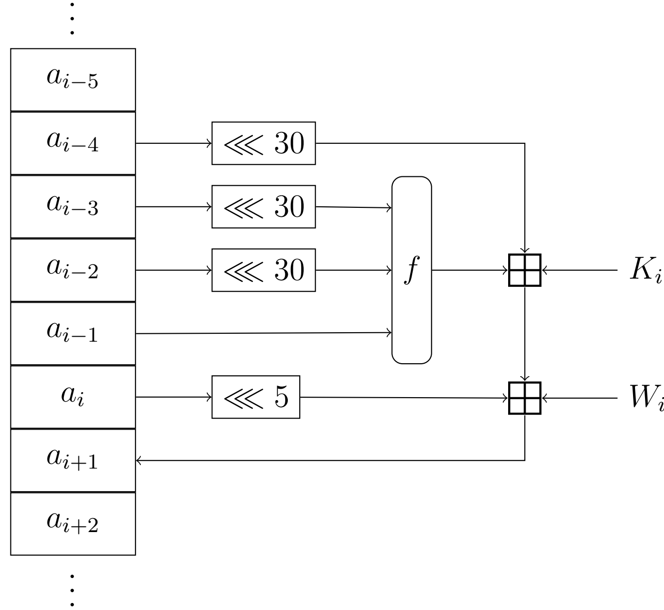


Figure 2.3: Alternative diagram of SHA-1's round function.

This function changes every 20 rounds and will be one of these:

$$f(x,y,z) = \begin{cases} IF(x,y,z) = (x \wedge y) \vee (\neg x \wedge z), & 0 \leq r \leq 19 \\ XOR(x,y,z) = x \oplus y \oplus z, & 20 \leq r \leq 39 \\ MAJ(x,y,z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z), & 40 \leq r \leq 59 \\ XOR(x,y,z) = x \oplus y \oplus z, & 60 \leq r \leq 79 \end{cases} \tag{2.3}$$

The *message expansion* relation for expanding the initial message words $W_0, \ldots, W_{15}$ from the 512 input bits to 32-bit message words for 80 rounds of SHA-1 is defined by

$$W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1, \text{ for } i \in \{16, \ldots, 79\}. \tag{2.4}$$

Equation 2.2 is the original description of the SHA-1 round function. We often use a more compact formulation of this function. The state words are labeled with a_is where b_i, c_i, d_i and e_i are represented with a_{i-1}, a_{i-2}, a_{i-3} and a_{i-4}. The round function gets as input $a_i \ldots a_{i-4}$ and outputs a_{i+1}. Figure 2.3 depicts this rounds function.

2.6.2 Description of SHA-256

SHA-256 is in the standard hash function family of SHA-2 [ErH11]. Its structure is similar to SHA-1, but with a more complex round function and message expansion. The input block size is 512 bits and it has 64 rounds. Using the following *message expansion* relation, the 16 32-bit input words, will be expanded to 64 32-bit words.

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, \text{ for } i \in \{16, \ldots, 63\}, \tag{2.5}$$

where

$$\begin{aligned} \sigma_0(x) &= (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3), \\ \sigma_1(x) &= (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10). \end{aligned} \tag{2.6}$$

The internal state is 256 bits consisting of eight 32-bit words labeled as $a_i, b_i, \ldots, h_i$ for each round i. In this document, we use a more compact alternative labeling, as many of these values, similar to SHA-1, are copied to next state words and only two words are updated. The *state update relations* is described with the following equations:

$$\begin{aligned} T_i &= E_{i-4} + \Sigma_1(E_{i-1}) + \mathrm{IF}(E_{i-1}, E_{i-2}, E_{i-3}) + K_i + W_i \\ E_i &= T_i + A_{i-4} \\ A_i &= T_i + \Sigma_0(A_{i-1}) + \mathrm{MAJ}(A_{i-1}, A_{i-2}, A_{i-3}) \end{aligned} \tag{2.7}$$

where

$$\begin{aligned} \Sigma_0(x) &= (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22) \\ \Sigma_1(x) &= (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25) \end{aligned}$$

and the functions IF and MAJ are the same as in SHA-1, K_i denotes the SHA-2 round constant, and W_i denotes the processed expanded message block.

2.7 Encoding SHA-1 and SHA-256 into SAT

A common method of encoding a function into a SAT instance is to use Tseitin transformation on the circuit implementation of the function. In this method, we introduce variables for inputs, outputs, and intermediate values. Each gate or block is translated into a CNF formula, which encodes $\phi : y \leftrightarrow f(x)$. Other encoding methods take an imperative approach where an explicit encoding for the higher level variables and constraints are given. In this work, we took the SHA-1 encoding of Nossum [Nos12] and tweaked it for our SHA-1 use cases and adapted it to generate instances for SHA-256. Our encoding is a mix of circuit and imperative encoding.

Basic blocks. Here we describe the encoding of building blocks in SHA-1 and SHA-256 described in sections 2.6.1 and 2.6.2. Other than the modular addition and rotation, all of the round-dependent logic functions are bitwise function. We need to apply the same operation for each bit in the 32-bit word, and hence the loop over bits in the encodings.

$$\begin{aligned} f = IF(x,y,z) &\equiv \bigwedge_{i=0}^{31} f_i \leftrightarrow (x_i \wedge y_i) \vee (\neg x_i \wedge z_i) \\ &\equiv \bigwedge_{i=0}^{31} (f_i \vee \neg x_i \neg y_i) \wedge (f_i \vee x_i \vee \neg z_i) \wedge \\ &\quad (\neg f_i \vee x_i \vee z_i) \wedge (\neg f_i \vee \neg x_i \vee y_i) \end{aligned} \tag{2.8}$$

$$\begin{aligned} f = XOR(x,y,z) &\equiv \bigwedge_{i=0}^{31} f_i \leftrightarrow (x \oplus y \oplus z) \\ &\equiv \bigwedge_{i=0}^{31} (\neg f_i \vee \neg x_i \vee \neg y_i \vee z_i) \wedge (\neg f_i \vee \neg x_i \vee y_i \vee \neg z_i) \wedge \\ &\quad (\neg f_i \vee x_i \vee \neg y_i \vee \neg z_i) \wedge (\neg f_i \vee x_i \vee y_i \vee z_i) \wedge \\ &\quad (f_i \vee \neg x_i \vee \neg y_i \vee \neg z_i) \wedge (f_i \vee \neg x_i \vee y_i \vee z_i) \wedge \\ &\quad (f_i \vee x_i \vee \neg y_i \vee z_i) \wedge (f_i \vee x_i \vee y_i \vee \neg z_i) \end{aligned} \tag{2.9}$$

$$\begin{aligned} f = MAJ(x,y,z) &\equiv \bigwedge_{i=0}^{31} f_i \leftrightarrow (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) \\ &\equiv \bigwedge_{i=0}^{31} (\neg f_i \vee x_i \vee y_i) \wedge (\neg f_i \vee x_i \vee z_i) \wedge (\neg f_i \vee y_i \vee z_i) \wedge \\ &\quad (f_i \vee \neg y_i \vee \neg z_i) \wedge (f_i \vee \neg x_i \vee \neg z_i) \wedge (f_i \vee \neg x_i \vee \neg y_i) \end{aligned} \tag{2.10}$$

Modular addition. The addition operations are modulo 2^{32}, therefore we just need to ignore any carry value beyond 32nd bit position. To encode these adders, one can encode a full-adder by generating CNF formulas of Majority function (for carry output) and 3-input XOR function (for sum output) using the equations 2.10 and 2.9, respectively, and then chain the full-adders together to create a ripple carry adder. Each round of SHA-1 has a 5-operand 32-bit adder, which can be encoded using four additions. Tseitin transformation of

ripple carry adder circuit, typically introduces a lot of auxiliary variables, and the chain of propagating information would be long. One way to optimize this is to instead of encoding multiple 2-operand additions, encode a multi-operand addition operation, to reduce the intermediate Tseitin variables.

Table 2.1: Applying column addition to multi-operand addition of five bitvectors.

c'_2	c'_1	c'_0		
	c_2	c_1	c_0	
	x_3	x_2	x_1	x_0
	y_3	y_2	y_1	y_0
	z_3	z_2	z_1	z_0
	t_3	t_2	t_1	t_0
$+$	w_3	w_2	w_1	w_0
$=$	s_3	s_2	s_1	s_0

Consider the following schema for a 4-bit 5-operand addition of x, y, z, t and w, and their sum s. The sum of five bits is a number between 0 and 5, thus it can be encoded as a 3-bit number. The least significant bit of this number goes to be the sum bit in that column, and the other two bits will be carried to the next columns (represented as $c'_i c_i$ in table 2.1). Each column receives at most two carry bits, so there are at most 7 input bits at each column, which is still representable with 3 bits (two carries and a sum bit). Therefore if we construct a counter function that accepts 7 bits of the same weight and adds them up and output a weighted 3-bit number, we can chain these counters (similar to ripple carry) to build a 5-operand addition ($f(x_i, y_i, z_i, t_i, w_i, c_{i-1}, c'_{i-2}) = c'_i c_i s_i$). Nossum [Nos12] proposed using a heuristic logic minimizer called ESPRESSO, to find the CNF formulation for such a function. ESPRESSO accepts the truth table of a Boolean function and gives out a minimized formula implementing that table. Our target function has 7 inputs, therefore it is feasible to enumerate the input possibilities and create the table. This encoding compared to a plain Tseitin encoding of SHA-1, reduces the number of variables by one fourth, at the cost of doubling the number of clauses. The resultant instances are shown to be easier to solve for CDCL solvers on average.

We implemented and analyzed different ways of encoding the multi-operand addition.

First, we encoded the counter function, using half-adders and full-adders, however, it was slightly worse than using Espresso. Next, we implemented the addition using Wallace matrix reduction that is commonly used in implementing multiplications. This method was on par with Nossum's method, and none of the two were significantly better than the other.

Consider a round reduced SHA-1 function $y = Hash(x)$ and let E be its CNF encoding. We can derive CNF encoding of related cryptanalysis problems by adding appropriate constraints to E. For the preimage of a hash value h, we need to force the output variables of $Hash$ to be equal to h using unit clauses and conjunct them with E $(E \wedge (y \leftrightarrow h))$. For encoding collision, we can have two copies of E with different Boolean variables and forcing the output variables to have the same value, while having at least one of the input variables to be different.

2.8 Algebraic Fault Attack

Implementation attacks. Implementation attacks are a type of attack on cryptographic primitives, where the attacker has access to an implementation of the cryptographic primitive (either on a hardware device or as a software running a computing platform), and the secret information is embedded within the implementation (or is a fixed input to the implementation). The attacker is able to query the implementation multiple times with chosen input messages (as plaintext to be encrypted or message to be tagged). There are two basic approaches to implementation attacks, namely, passive and active implementation attacks. In passive attacks, the attacker measures some aspect of the computations on a target implementation via side-channel such as power consumption or timing, to find patterns that can be exploited. By contrast, in active attacks, the target implementation is manipulated as part of the attack. In this dissertation, we consider only active attacks. *Fault injection analysis* is a form of active attacks, where the attacker intentionally introduces faults in the operation of cryptographic function and analyzes the incorrect outputs to recover the embedded secret key. These faults could be injected via a variety of methods, like heating or varying the voltage of the power supply in a controlled fashion to attack hardware implementing these functions [ADN+10, BECN+06, BBKN12]. Fault attacks were first proposed in 1997 as a way to break RSA-CRT cipher (cf. [BDL97]). There are broadly two classes of fault attacks that researchers have studied, namely, *differential fault attacks* (DFA) and *algebraic fault attacks* (AFA).

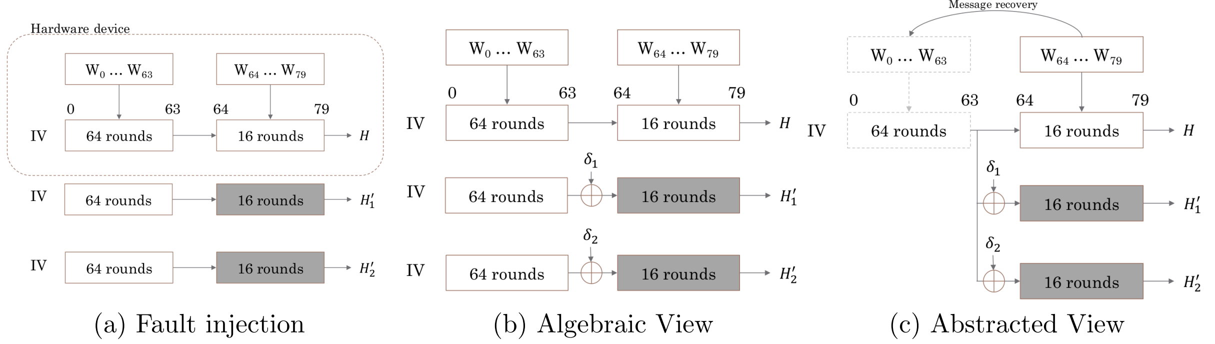
Differential Fault Attack. The DFA method was first proposed for breaking DES cipher [BS97] and has been applied to many other block ciphers [AMT13, JL12, LLG09], stream ciphers [HR08], and hash functions [HH11, FR12, LFZD16]. At a high level, the DFA method exploits the differences in the relation between the faulty outputs and the intermediate variables compared to the correct outputs in order to recover an inner state. Propagation of induced faults in the forward direction and deduction of fault differences, backward from the output to the fault location, so-called *fault equations*, is examined manually by a cryptanalyst.

Algebraic Fault Attack. AFA methods combine fault injections with algebraic cryptanalysis [CJW10]. In this approach, the cryptographic function and faults are translated into algebraic equations over a finite field, and the secret key or message is recovered by solving these equations using a SAT or SMT solver. Fault equations refer in this case to an algebraic representation of the cryptographic function starting from the injected fault location up to its output. The advantage of AFA over DFA is that the solver takes care of propagation of the fault, and thus significantly reducing the human effort required to launch a successful attack. AFA has been used to automate DFA methods on block ciphers [ZGZ+13, ZZG+13], stream ciphers [MBB11a] and hash functions [HLMS14, LAFW17]. Figure 2.4 shows a high level view of AFA on SHA-1 hash function. Figure 2.4a depicts injection of faults on a hardware implementation of SHA-1, where the faults are induced before the last 16 rounds, and causing the device to output a faulty value.

Figure 2.4b shows how the faults are seen in an algebraic view, where all cryptographic relations are encoded as algebraic equations and each injected fault is encoded by XORing a random and unknown value to the variable corresponding to the fault location. It can be seen that all of the unfaulty equations (from input to fault location) are just a repetition of the original SHA-1 relations. Figure 2.4c shows how these duplicated relations are abstracted away and the remaining parts are the ones that actually get encoded into AFA equations. Because we are abstracting equations, after solving the equations, we need to recover the actual messages and verify whether the solution is spurious or not.

2.9 Differential Cryptanalysis

Broadly speaking, differential cryptanalysis [BS91] is the analysis of how a difference in the input values of a cryptographic function can affect the resultant difference at the output. Block ciphers and cryptographic hash functions are typically comprised of chaining of



(a) Fault injection (b) Algebraic View (c) Abstracted View

Figure 2.4: The hardware fault injection in the input of last 16 rounds of SHA-1 and the algebraic encoding of faulty runs.

smaller functions. In these cases, differential cryptanalysis looks at the trace of differences of values through the chain of transformations to find non-random behaviors of the function and exploiting it to find input messages or secret keys.

For systems defined over finite field of characteristic 2, the difference is often defined with the XOR operation ($\Delta x = x \oplus x'$). We are interested in relations between Δx and $\Delta y = f(x \oplus \Delta x) \oplus f(x)$, for a cryptographic function f. The differential probability of a vectorial Boolean function is defined as follows:

Definition 2.9.1. Let $\Delta x \in \mathbb{F}_2^n$ be the input difference and $\Delta y \in \mathbb{F}_2^m$ be the output difference. For a vectorial Boolean $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$, the differential probability of $\Delta x \to \Delta y$ is defined as:

$$dp(\Delta x \to \Delta y) = \frac{|\{x \in \mathbb{F}_2^n | f(x) \oplus f(x \oplus \Delta x) = \Delta y\}|}{2^n}$$

Difference distribution table (DDT) is a table that contains the number of pairs for all input/output differences. For functions with a small domain size (e.g. an S-box) the DDT can be efficiently populated. For large block ciphers and hash functions it is infeasible to generate the DDT. However, block ciphers and hash functions are often iterative functions that are build by applying small round functions repeatedly. Therefore the differentials over the smaller steps of f is analyzed and chained together to derive differentials over input/outputs of f. This trail of differentials is called a *differential characteristics*:

Definition 2.9.2. For an r-round iterative function $f = f_{r-1} o \cdots o f_1 o f_0$, a sequence of differences

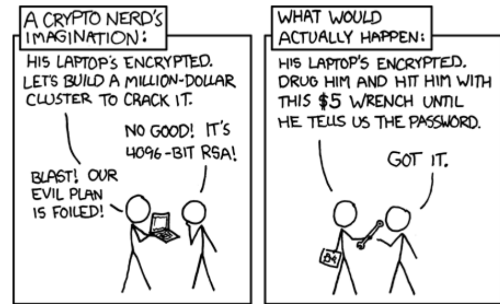
$$\Omega : \delta_0 \xrightarrow{f_0} \delta_1 \xrightarrow{f_1} \delta_2 \to \cdots \to \delta_{r-1} \xrightarrow{f_{r-1}} \delta_r$$

is called an r-round differential characteristics of f. For a differential $\Delta x \to \Delta y$ over f, we have $\delta_0 = \Delta x$ and $\delta_r = \Delta y$.

2.10 Terminology

Cactus plots. Most of the plots presented in this document are cactus plots. In these types of plots, each data point (X, Y) shows that X instances are solved under Y seconds. This means that solvers that are further to the right are solving more instances and solvers that are further to the bottom are solving instances faster.

PAR-k. Penalized Average Reward, is an evaluation measure that is being used in recent SAT competitions to rank SAT solvers. We also use this metric in some of our experiments to compare solvers on a specific benchmark. PAR-k of solver S on a benchmark B is the sum of runtimes of S on all instances in B, counting each timeout as k times the running time cutoff. In the competition and in our experiments $k = 2$ is used.



XKCD(https://xkcd.com/538/)

Chapter 3

CDCL(Crypto) SAT Solvers

Boolean satisfiability (SAT) solvers are well-known powerful general purpose search tools, that have been used in solving problems from many different domains, such as verification, AI, and cryptography [CGP+08], [Rin09], [MZ06]. They get their power from reasoning components like clause learning [MSS99] and many different search heuristics, like VSIDS or machine-learning based LRB branching [MMZ+01], [LGPC16a] clause deletion [AS09b] and restarts [AS12].

SAT-based Cryptanalysis. The availability of such powerful search tools has led many researchers to propose the use of SAT and SMT solvers for cryptanalysis of hash functions and symmetric encryption schemes, for example in preimage attacks [MS13], [Nos12], collision attacks [MZ06], [Pro16] and linear and differential cryptanalysis of block ciphers[ADWL17], [KLT15].

Although in some of the approaches, the heuristics of the solver are altered to improve their efficiency, e.g. branching heuristics [Pro16], [SZBP11] and restart policy [NLG+17], most of these approaches used a direct encoding of the said problems into a satisfiability problem and used SAT solvers as a blackbox, and the changes are limited to the search heuristics and do not alter the logic reasoning components of the solver. The one notable exception is the CryptoMiniSat solver [SNC09], that adds reasoning over XOR clauses to the solver to improve the solving of cryptographic instances that heavily use XOR operations.

The current work on SAT-based cryptanalysis is similar to the *eager* approach in solving Satisfiability Modulo Theories (SMT) formulas, where the formula is directly translated into a SAT instance and then a SAT solver is invoked on it. The benefit of this approach is that we can use any SAT solver as-is and leverage the performance of the solver and

its improvement capacity over time. The downside of this approach is the loss of the high level semantics of the underlying theories. This means that the SAT solver needs to perform extra computations to prove facts that are readily available in the higher level logic (e.g. $x + y = y + x$ in the integer arithmetic). The other main approach of solving SMT instances, called *lazy* approach, integrates the CDCL style search with theory-specific solvers (T-solvers). This architecture is referred to as CDCL(T). Generally speaking, a T-solver is useful only if it participates in propagation and conflict analysis reasoning engines of the SAT solver they extend.

3.1 Contributions

The main research question that we pose in this chapter is:
Q: Are there methods that can surpass blackbox SAT-based cryptanalysis in terms of scalability and ability to break complex real-world cryptographic primitives?

1. **CDCL(Crypto) framework.** Inspired by the CDCL(T) paradigm, we propose a framework for SAT-based cryptanalysis that we call CDCL(Crypto). It extends the propagation and conflict analysis of the core SAT solver using the higher level domain knowledge about the cryptographic problem that is being analyzed. To be more flexible, and to have simpler implementation and be able to customize the extended functionalities to different cryptographic problems, we use the *Programmatic SAT* [GOS+12] architecture, where the solver provides callbacks for extending propagation and conflict analysis to be implemented by the user.

2. **Case study 1: Algebraic Fault Attack.** We first review an application of this framework that has been successfully applied to algebraic fault analysis of SHA-1 and SHA-256 cryptographic hash functions [NHGG18], enabling the attacker to recover the secret bits with only 11 faults in SHA-1 and 48 faults in SHA-256, which is a significant improvement over previous algebraic fault attacks.

3. **Case Study 2: Differential Cryptanalysis.** Then we demonstrate that this framework can be applied to other cryptographic problems, more specifically differential cryptanalysis of round reduced SHA-256. We present preliminary results on increasing the number of rounds in the collision finding of SHA-256 compared to the previous SAT-based differential cryptanalysis of SHA-256.

3.2 CDCL(Crypto) Framework

In this section, we describe the CDCL(CRYPTO) framework, based on a programmatic SAT solver, illustrated in Figure 3.1.

3.2.1 Programmatic Interface in SAT Solvers

We call a SAT solver *programmatic* [GOS+12] if it is augmented with a set of callback functions that allow the user to add functionality to the solver's propagation and conflict analysis routines. The idea is inspired by the CDCL(T) architecture, in which a theory solver provides support for theory propagation and theory conflict analysis to the base Boolean CDCL solver. Programmatic SAT solving differs from the general concept in 3 ways: First, the theory solver in the context of programmatic SAT can be an arbitrary piece of code, in that we place no requirements on its completeness; second, this code might be particularized to every input to the solver. That is, unlike the T-solver in CDCL(T) which remains invariant for all formulas from the language of T, the code added via the programmatic interface in a programmatic SAT solver can be specific and unique to each input; and finally, the interface of programmatic SAT solvers is much simpler than that of SMT solvers.

The main advantage of using programmatic SAT is that it allows easy customization of the SAT solver to specific Boolean instances rather than an entire theory. The developer thus has more fine-grained control over the power of the SAT solver. This architecture has also shown to be useful in solving problems in combinatorics [BGH+16], and much more effective than only using a normal CNF encoding. Figure 3.1 shows the block diagram of a CDCL SAT solver and the connection of programmatic components (shaded blocks) to the main components.

Programmatic propagation has the role of providing clauses similar to theory propagation clauses. As can be seen in the figure, there is a close interaction loop between unit propagation and programmatic propagation, in which when the unit propagation is done, if there is no conflict, programmatic propagation analyzes the partial assignment and determines whether any other literal is implied according to the logic of the cryptographic function. If any literal is implied but missed by the unit propagation, an appropriate *reason clause* is returned to empower the unit propagation. Consider that α is a subset of literals in the partial assignment that implies another literal L, and this implication is missed by unit propagation. The added reason clause will be simply $\alpha \to L$ (in CNF format). Then

the unit propagation is invoked to set those literals and possibly find more implications that are caused by the new literals. Added reason clauses can be reused when the solver unassigns some of the variables and assigns them again (due to backjump or restart).

Programmatic conflict analysis in a similar fashion, is invoked when the propagation is done (a combination of unit and programmatic) and no conflict is detected. It analyzes the partial assignment to check if there is conflicting information according to the domain knowledge. The user can return single or multiple conflict clauses if a conflict is detected. The core solver then looks at the variables that are in the conflict clause, and by examining the implication graph that has been built during the run of the solver, attempts to find a minimized root cause of the conflict.

We have implemented this framework on top of MapleSAT [LGPC16a]. Programmatic routines need to know the mapping of the high level variables to the Boolean variable IDs. This is necessary to be able to verify the value of a predicate when the corresponding Boolean variables are set. In order to keep the variable ID mapping intact, we do not use any pre-processing that re-indexes the variables. During the search, the size of the conflict clause database only increases and this might negatively impact the performance of unit propagation. To handle this challenge, modern SAT solvers regularly delete some of the lower quality clauses. In the programmatic SAT, the same problem could happen for the reason clause database. In our implementation, we use the same clause deletion strategy of MapleSAT to prevent the overgrowth of the reason clause database.

3.2.2 Cryptographic Reasoning in Programmatic Callbacks

Even for cryptographic functions that use very simple operations, like addition-rotation-xor (ARX) block ciphers and hash functions, some high level properties like commutativity of addition, is lost when translated into the Boolean level, let alone much more complex cryptographic properties. One can specifically encode these properties, but it will result in a very large SAT instance (e.g. commutativity of multi-operand additions in ARX). The programmatic approach enables us to express those properties concisely using a piece of code (C++ in our case), that are being used by the SAT solver through the programmatic interfaces. We will give more detailed use of these interfaces in two cryptanalysis applications. In section 3.3, we present an algebraic fault attack on SHA hash functions [NHGG18] and also present preliminary results on differential cryptanalysis of SHA-256 in section 3.7.

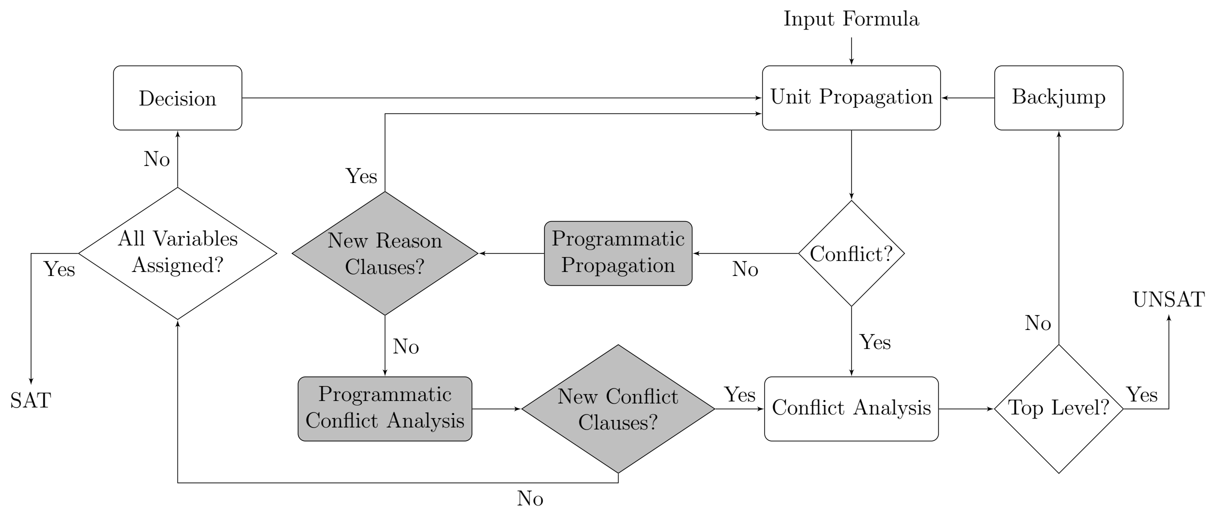


Figure 3.1: Block Diagram of a CDCL SAT solver with the Programmatic components that implement cryptographic related reasoning (shaded blocks).

3.3 Algebraic Fault Attack

Cryptographic hash functions, such as the SHA family, play a critical role in a variety of settings in cryptography (e.g., authenticated encryption, pseudo random number generation, digital signatures, etc.) [MVOV96]. While there is some recent progress on practical collision attacks on SHA-1 [SBK+17], inversion attacks on the full version of the standard SHA family of functions are still impractical [DEM14]. Given that these functions seem highly resistant to direct inversion attacks, many researchers have turned to *implementation inversion attacks*, wherein, the attacker gathers information on implementations of these hash functions (or any cryptographic primitive) in an attempt to reduce the size of search space. One form of this type of attack called *fault injection analysis* involves intentionally introducing faults in the operation of cryptographic devices and analyzing the incorrect outputs to recover the embedded secret key.

Algebraic fault attack methods combine fault injections with algebraic cryptanalysis [CJW10]. In this approach, the cryptographic function and faults are translated into algebraic equations over a finite field, and the secret key or message is recovered by solving these equations using a SAT or SMT solver. Fault equations refer in this case to an algebraic representation of the cryptographic function starting from the injected fault location up to its output. The advantage of algebraic fault attack over differential fault attack is that the solver takes care of propagation of the fault, and thus significantly reducing the human

effort required to launch a successful attack.

AFA methods are a powerful way of empirically verifying the strength of cryptographic function's implementation through fault analysis. AFA methods have significant advantages over previous approaches since they leverage the continuous scalability improvements in SAT and SMT solvers. Having said that, it is well known that merely using the solver as a blackbox (à la the *eager* approach) is not going to yield the best results. The efficacy of AFA methods broadly relies on three important factors that any SAT/SMT solver user would readily recognize, namely, the type of encoding of the cryptographic primitive in Boolean or suitable SMT logic, the underlying solver, and the effectiveness with which the user is able to tune or modify the underlying solver's heuristics.

In their original paper on AFA [CJW10], the authors describe a *lazy* approach to AFA, wherein, part of the cryptographic primitive (more precisely, the fault-injected part) is translated into a Boolean formula, and the rest of the primitive is used to verify solutions generated by the solver. If the solution is incorrect, their tool blocks it by adding an appropriate clause to the solver and repeats until the correct solution is found. While their method is clearly sound, complete, and terminating, the authors do not exploit the solver's power in a whitebox fashion nor do they explore encodings that may be best suited for an algebraic fault attack. While researchers have explored different kinds of encodings subsequent to the paper by Courtois et al. [CJW10], none of them use the underlying SAT/SMT solver in CDCL(T) fashion (cf. [ZGZ+13, ZZG+13, HLMS14]).

3.3.1 High-level Overview of Our Method to Algebraic Fault Attack

In this work, we propose a programmatic SAT solver-based method [GOS+12] for AFA, wherein, we extend both the *Boolean constraint propagator* (BCP) and the conflict-analysis in a state-of-the-art SAT solver, MapleSAT [LGPC16b]. Our extension of BCP is similar to theory propagation in CDCL(T), and we refer to this extension as the SHA propagator. The conflict-analysis extension is similar to the theory conflict analysis in CDCL(T), and we refer to this extension as the SHA conflict analyzer.

How our Method Works. At a high level, in our method, the fault-injected part of the hash function, along with a target, is translated into a Boolean formula (which is then fed as input to the SAT solver), while the full implementation of SHA is encoded via a programmatic interface as part of the SAT solver's propagation and conflict analysis

routines. Such an approach enables the addition of conflict clauses to the appropriate database in an on-demand and lazy fashion. We refer to our addition to the solver's propagation routine as the *SHA propagator*, and the one to the solver's conflict analysis routine as the *SHA conflict analyzer*. We evaluate our tool under a variety of fault models and show that we can recover the secret bits (in our setting, the secret bits correspond to any message that hashes to the given target) with a fewer number of injected faults compared to previous best work (reducing the cost of attack). Although fault injections are done on hardware devices, in this work we are simulating the fault injection process in software, by picking a random value as embedded secret bits and XORing random values to the intermediate state words as fault injection.

SHA Propagator. While analyzing different encodings of the SHA hash functions in Boolean logic, we noticed that the native BCP in SAT solvers does not propagate all the input bits (once set) all the way to the output bits of the SHA function for certain kinds of encodings of SHA in Boolean logic. More precisely, given a Boolean function f over input variables x and output variables y, there exist encodings ϕ_f (in conjunctive normal form) such that the standard-issue BCP does not propagate the values assigned to x all the way to y. In other words, the encoding ϕ_f is not preserving the generalized arc consistency. A natural and cost-effective way to strengthen the native BCP in SAT solvers would then be to add a SHA propagator that propagates inputs to the encoding of SHA to all its output bits, and adds clauses to the clause database appropriately. In our experiments, this method alone gave a massive boost to the performance of MapleSAT over AFA instances.

SHA Conflict Analyzer. As alluded to above, the SHA conflict analyzer is the checker that verifies whether the solution found by the solver is indeed a valid message for the given target. If not, a conflict clause is added to the conflict clause database of the solver. This mechanism prunes the search space dramatically according to our experiments. Otherwise, the validated solution is output by the solver. Unlike the AFA method proposed by Courtois et al. [CJW10], our SHA conflict analyzer is called in the inner loop of the SAT solver thus taking advantage of both its inherent incrementality and conflict analysis capabilities.

3.4 Programmatic Callbacks for Algebraic Fault Attack

In this section, we explain what is the role of each callback in the context of our programmatic SAT solver solving algebraic fault analysis equations.

3.4.1 Programmatic Conflict Analysis

We are only interested in the values of message bits, which are a very small subset of all of the variables needed to encode the algebraic fault equation system into CNF. Whenever we solve the instance and find the message bits, we should check if it is a legitimate solution (hashes to the same correct hash output). Normally one could wait for the solver to finish solving the whole equation set and then check for the correctness, but we can do this verification as soon as the variables corresponding to the message bits are set. The sooner we reject a spurious solution, the faster the search process becomes. The programmatic conflict analysis callback is invoked when the solver's Boolean propagation routine reaches an inconclusive state or all of the variables are assigned, and there is no conflict. First, it recovers the original input message bits, if all message bit variables are set, then hashes the input message bits and checks it against the correct hash output. In case of mismatch, a conflict clause that blocks the current spurious message bits will be returned to the solver. The solver has the reason clauses that led to this partial assignment, thus it can further optimize the returned clause using the implication graph, which makes the blocking clause more effective. The solver then goes through the procedure of backjumping, as in the typical conflict analysis.

3.4.2 Programmatic Propagation

It is known that when encoding a problem into CNF, we might lose some structural information about the original problem. For example, setting a subset of variables in a CSP instance might imply the value of another variable. But if the encoding of that CSP problem into CNF is not UP-maintaining GAC, then by setting the corresponding variables in the Boolean formula, BCP may not be able to derive the value for the target variables. An example of such an encoding is listed in Example 2.3.1 in Section 2. It is also mentioned in [PS15] and [ES06] that encoding of a pseudo-Boolean constraint into CNF using adder networks does not maintain GAC, although these encodings are small and scalable. To

overcome this problem, one might use arc-consistent encodings for a particular constraint or use enhanced propagation routines, e.g., bitvector propagators [WSS16].

In this work, we deal with cryptographic functions having multi-operand additions in each round. There are several encodings for these operations in the literature. Nossum's encoding [Nos12] gives a very compact CNF, which works very well in practice. Unfortunately, a curious side effect of having this minimal encoding is that after setting all of the input bits, BCP might not be able to set all output bits. There are two options to work around this problem, either empower the encoding or strengthen the unit propagation. Based on our initial experiments and experiments in [Nos12] with straightforward Tseitin encoding of adders, empowered encoding of adders can become very expensive (reaching time limit on all instances). Better propagation (based on SHA-1) would be effective no matter the encoding. We, therefore, explored the latter option in this work.

Our programmatic propagation (PP) is called in the main search loop of the solver after BCP is done, and no conflicts are detected. The callback looks at the least significant bits of the operands and output in each of the multi-operand additions. If all bits up to some bit position k are set, it checks if the k least significant bits of the output are set as well. If they are not set, it returns clauses that encode the direct implication between input bits and output bit in the missing output bit positions. For an example of encoding implications, if $x = T$, $y = F$ is an assignment to the inputs of $z = x + y$ relation, and z is not set, we return $x = T \wedge y = F \rightarrow z = T$ or $\neg x \vee y \vee z$. These implications force the solver to set the output bits in the next cycle. Although our implementation finds more implications than unit propagation does, it is not guaranteed that every encoding PP-maintains GAC according to Definition 2.3.1.

Definition 3.4.1. Let ϕ be a CNF encoding of a Boolean function f, and let R be an inference rule of propositional logic. We say that ϕ R**-maintains Input/Output GAC**[1] if for an assignment α that contains assignments to the input variables of f, the assignment of the output variables of f are derived from $\phi \wedge \alpha$ by R.

For example, a direct Tseitin encoding of a CIRCUIT-SAT instance to CNF has the property given in Definition 3.4.1. Our implementation of programmatic propagation looks at the inputs of the multi-operand addition and generates direct implications between input and output bits. If any subset of the input bits is set, and a subset of the output bits can be determined (through addition), those output bits are set either by unit propagation through formula clauses or through the direct implication clauses. Therefore we can say that any CNF encoding of multi-operand addition PP-maintains Input/Output GAC.

[1]GAC refers to Generalized Arc-Consistency defined in Definition 2.3.1.

3.5 Algebraic Fault Attack on SHA-1 and SHA-256

Here we describe how the attack is mounted on the SHA-1 and SHA-256 compression functions, and where programmatic callbacks fit in. For encoding of SHA in CNF, we used an adapted version of Nossum's encoding [Nos12], which is described in Section 2.7.

3.5.1 Algebraic Fault Notations

In practice, faults are induced on a hardware implementation using a device that can generate perturbation, e.g., radiation, heat, laser, etc. The attacker chooses a specific register and applies the fault, which changes the input to the subsequent operations. The choice of which register to apply fault is important, and we refer to that register by *fault location*. The change to the targeted register's value is usually unknown. But with more sophisticated (and more expensive) devices it is possible to narrow down the number of bits in the state that the fault injector is affecting. Therefore the number of bits that can be flipped is a parameter that represents how strong is the attacker. The number of flipped bits shows the *hamming weight of the fault vector* applied and is usually referred to as the *fault model*. Another parameter in our AFA model is the *number of faults* that the attacker is capable of injecting. This parameter represents the cost of the attack, and thus the fewer injections the better.

In the algebraic setting, the transformations from the fault location to the output are encoded as constraints (in our case in CNF), and we refer to it as *correct equations*. For each injected fault, the transformations from the fault location to the output are again encoded but the output value is fixed to the corresponding faulty output, and we refer to them as *faulty equations*. The variables corresponding to the secret message bits are shared between all of these equation sets. Depending on the device that is used for fault injection, the attacker can assume an upper bound on the hamming weight of the difference between correct and faulty values of the fault location register. This can also be encoded as a constraint.

3.5.2 Attack Model

We assume that the attacker picks and knows the location of the fault, but does not have control over the value of the fault. We also assume that the chaining value at the input of the compression function is fixed to the initialization vector. Note that we do

Algorithm 2 `AFA-SHA` (An Algebraic Fault Attack on SHA)

Input: f: a SHA compression function, g: a reverse message expansion of SHA, d: the maximal weight of faults, L: a list of fault locations, k: the number of faults (k is divisible by $\#L$), H: the correct SHA hash image.

Output: M': a message, such that $f(M') = H$.

1: **function** `AFA`(f, g, d, L, k, H)
2: Let M represent the embedded secret SHA message
3: Let n be the number of rounds in f.
4: Let ϕ be a CNF encoding of $H = f_{(n-15)..n}(x)$.
5: $\Phi := \phi$
6: **for** ℓ **in** L **do**
7: **for** $i = 1, \ldots, k/\#L$ **do**
8: Generate a random fault value δ_i with $w_H(\delta_i) \leq d$.
9: $H'_i := f_{(\ell+1)..n}(f_{1..\ell}(M) \oplus \delta_i)$ ▷ Calculate the faulty output
10: Let ϕ_i be a CNF encoding of $H'_i = f_{(\ell+1)..n}(x \oplus \delta_i)$.
11: $\Phi := \Phi \wedge \bigwedge \phi_i$.
12: **repeat**
13: Find a model α for Φ.
14: Extract the assignment for $W_{n-15}, \ldots, W_n$ from α.
15: **for** $j = n - 16, \ldots, 1$ **do**
16: $W_j := g(W_{j+1}, \ldots, W_{j+16})$
17: $M' := W_0 \| \ldots \| W_{15}$
18: $\Phi := \Phi \wedge \neg M'$
19: **until** $f(M') = H$
20: **return** M'
21:
22: **function** `g-SHA-1`$(W_0, \ldots, W_{15})$ ▷ SHA-1 Message expansion in reverse
23: **return** $((W_{15} \ggg 1) \oplus W_{12} \oplus W_7 \oplus W_1)$ ▷ see Equation 2.4
24:
25: **function** `g-SHA-2`$(W_0, \ldots, W_{15})$ ▷ SHA-256 Message expansion in reverse
26: **return** $(W_{15} - \sigma_1(W_{13}) - W_8 - \sigma_0(W_0))$ ▷ see Equation 2.5

not perform actual hardware fault injections and the process is simulated in software by XORing random values to the inner state variables.

3.5.3 Attack on SHA-1

In our attack described in Algorithm 2, we target the last 16 rounds of SHA-1. The message expansion is invertible, provided we have 16 consecutive words (see Equation 2.4). This means that recovering the last 16 expanded message words enable us to recover all message bits. Therefore we inject faults to the input of the last 16 rounds, and more particularly in b_{64}. This fault location is more desirable because of the way the fault propagates in the next rounds. For more details, we refer to [HH11].

Let f be the compression function of SHA-1. Let $f_{1..64}$ (resp. $f_{65..82}$) be the Boolean map representing the first 64 rounds of f (resp. the last 16 rounds of f). Thus we have the following composition $f = f_{65..80} \circ f_{1..64}$. Let M be a SHA-1 message. Consider the correct hash value $H = f(M)$. We can encode fault outputs as $H'_i = f_{65..80}(f_{1..64}(M) \oplus \delta_i)$, where δ_i is a random fault value. These are the steps that we follow:

- We obtain the correct hash output H and several faulty outputs H'_i for the given M.
- Then we encode the set of correct and faulty equations for the last 16 rounds in CNF. Figure 3.2 shows the parts of the compression function that are being encoded into CNF.
- The composed formula Φ is then given to the SAT solver to find a solution for the last 16 message words.
- The verification loop is implemented in the SHA conflict analyzer. As soon as the corresponding variables to $W_{65}, \ldots, W_{80}$ are set, the analyzer will derive the first 16 message words M' by applying the Equation 2.4 in reverse (see `g-SHA-1` in Algorithm 2). This value is given to the compression function to see if it hashes to H. If there is a match, the found M' is the final solution, otherwise, the last 16 message words will be returned as conflict clauses to the SAT solver, and the search loop continues.

The attack is run by calling the `AFA` function from Algorithm 2 with the following arguments: `AFA`(f_{SHA1}, d, `g-SHA-1`, L:{64}, k, H), where f_{SHA1} is the SHA-1 compression function.

3.5.4 Attack on SHA-256

Our attack on SHA-256 shares the same framework as in the SHA-1 attack. Our approach is outlined in Algorithm 2. Just like in the SHA-1 attack, we target the last 16 rounds of

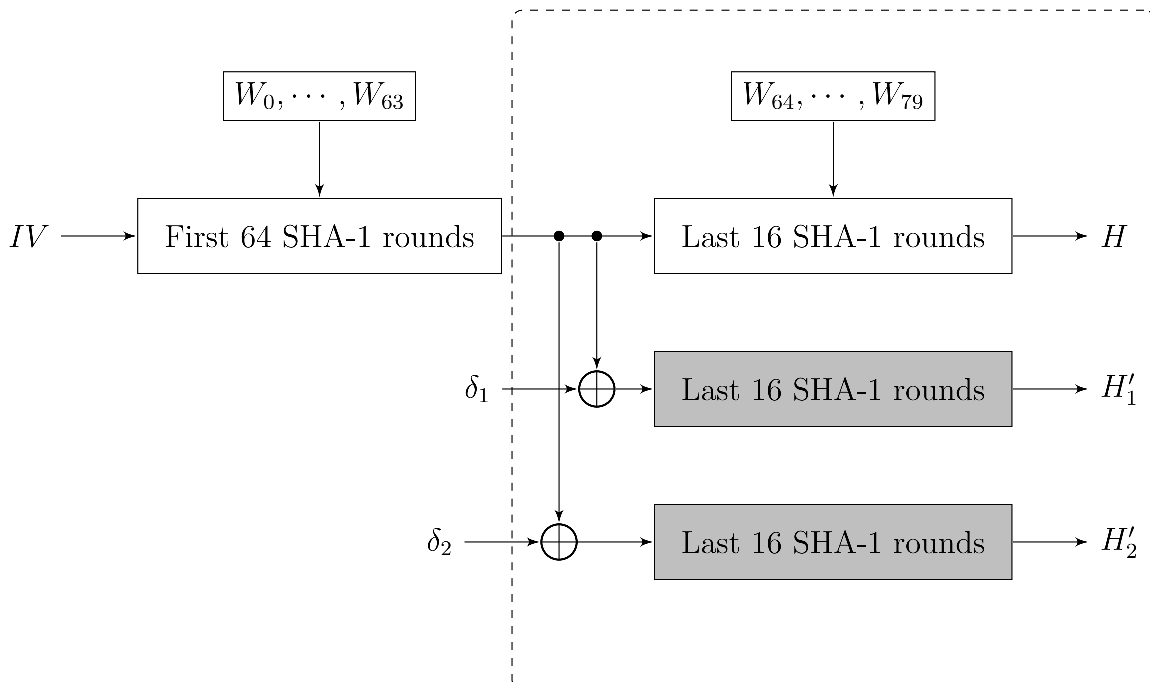


Figure 3.2: A high-level diagram of the SHA-1 attack. The values δ_1 and δ_2 represents the injected faults. H denotes the correct hash output and H'_1 and H'_2 are the faulty outputs. The dashed box is the part that is being encoded into CNF. The shaded boxes are copies of the white 16 rounds, and $W_{64}, \cdots, W_{79}$ variables are shared between all of them.

SHA-256 and the deepest fault location is c_{48}. For details on the impact of choosing this location, we refer to [HLMS14]. Since the state update operations in SHA-256 are more complex, the size of encoding is much bigger and the instances are harder to solve. We set a higher time limit and use a multi-stage fault injection approach to limit the effect of fault propagation.

Hao et al. [HLMS14] presented an AFA on SHA-256. They first target the last four rounds, inject faults and solve the equations to recover $W_{61}, \ldots, W_{64}$. Then they fix the message words to the found solution and repeat the same procedure for the next four message words. This means that with another set of fault injections, they recover $W_{57}, \ldots, W_{60}$, and so on, to find the last 16 message words. We follow the same approach to keep the size of instances small. An immediate challenge in this approach is to check the consistency of the solutions for each set of four message words with the hash values. In our approach when we encode all of the relations from the fault injection location to the output, in-

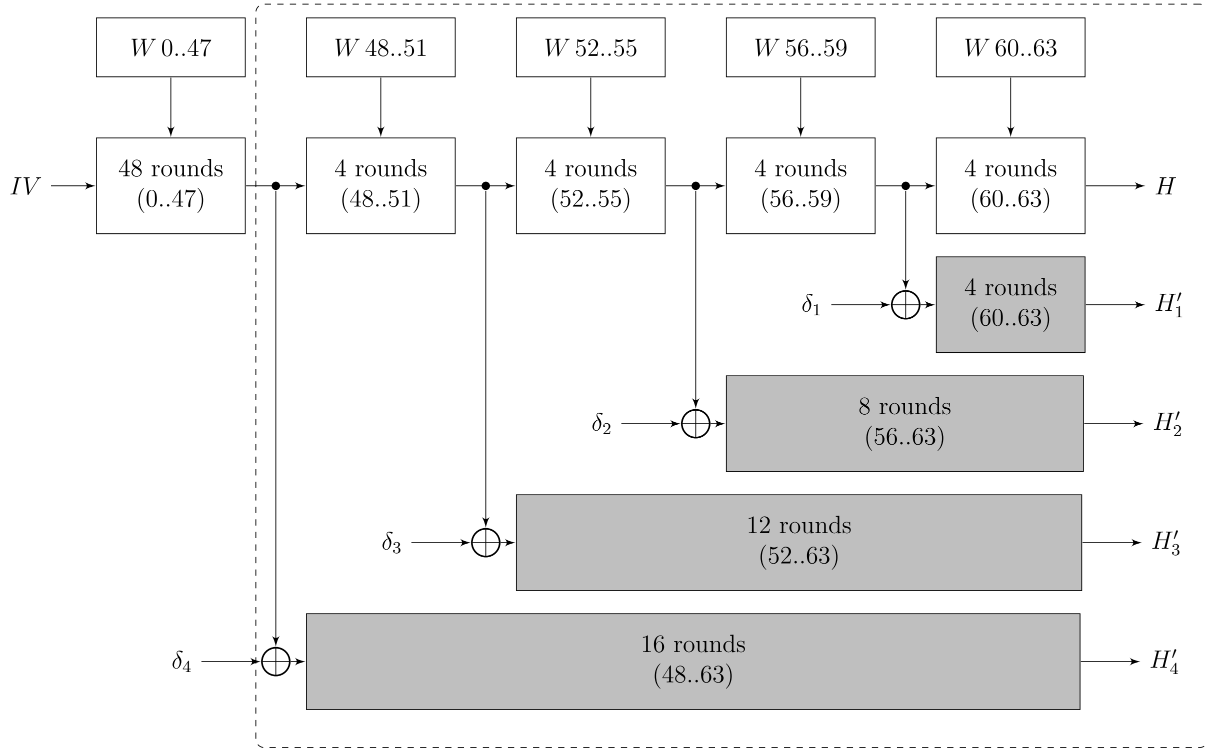


Figure 3.3: A high-level diagram of the fault attack on SHA-256.

stead of solving the instances in each step and fixing the solution in other instances, we conjunct all of the encoded instances together and let the solver handle the consistency of solutions. Following this method, we target the last four message words by injecting faults in round 60 and encoding the fault equations. Next, we inject faults in round 56 to target the last 8 rounds. Similarly, we target the last 12 rounds and the last 16 rounds. All of the encoded fault equations together with the correct hash function relations for the last 16 rounds make our SAT instance. We inject the same number of faults in each of those fault locations. Similar to SHA-1, the verification process is implemented in the SHA conflict analysis callback, with the difference of using Equation 2.5 (see `g-SHA-2` in Algorithm 2), for deriving the first 16 words of M' and SHA-256 compression function is applied to check with the correct output H. Using the `AFA` function from Algorithm 2, the attack is launched with this call:
`AFA`(f_{SHA2}, d, `g-SHA-2`, L:{60, 56, 52, 48}, k, H), where f_{SHA2} is the SHA-256 compression function.

3.6 Experimental Results

3.6.1 Experimental Setup

All experiments were conducted on Intel Xeon CPUs at 3.2 GHz and 16 GB of RAM. We used MapleSAT [LGPC16b] to implement the programmatic callbacks. There are other SAT solvers like CryptoMiniSAT and lingeling that implement XOR reasoning which could be beneficial in solving ARX[2] cryptographic functions like SHA-1 and SHA-2. Also, SMT solvers that handle bitvectors, like STP, are a good candidate in solving these kinds of instances. But in practice, according to the study in [NLG+17], MapleSAT outperforms them on SHA-1 preimage instances. Because of the similarity of SHA-1 preimage instances to our fault instances we picked the best solver and implemented our programmatic interface in it. We have also decided to use the multi-armed bandit restart (MABR) policy [NLG+17] in MapleSAT, which adds an additional performance gain on cryptographic instances. We experimented with various assumptions on the number of the injected faults and on the maximal weight of the faults. For each experiment, we generated 100 random message-target pairs, and the timeout was set at 4 hours for SHA-1 instances and 12 hours for SHA-256 instances. For the sake of completeness and fair comparison, we have added an external loop around MapleSAT that does the verification (repeat-until loop in Algorithm 2) and adds blocking clauses to the solver if an inconsistent solution is found. In this section, whenever we mention the base version of MapleSAT, we mean MapleSAT with the verification loop.

3.6.2 Attack on SHA-1 and SHA-256

Table 3.1 shows the results of applying AFA on SHA-1 and SHA-2. Its rows correspond to the maximal weight of the injected faults. Its columns correspond to the number of injected faults during the attack. Starting from a single bit, going to a nibble, a single byte, single word, and the most relaxed one is the 32-bit random fault model. Each element in Table 3.1 represents the number of instances out of 100 randomly generated AFA instances that our solver was able to solve within the time limit. From Table 3.1a we can see that we are able to recover the message bits with as few as 8 faults in the single byte fault model. In previous attacks on SHA-1, Hemme et al. [HH11], apply a DFA that uses 1002 faults. In the same fault model (32-bit fault model), we use only 11 faults.

[2] Addition-Rotation-XOR

Table 3.1: The number of solved AFA instances out of 100 for different number of faults and maximal weight of the faults

(a) SHA-1

Fault weight \ Number of faults	8	11	12	16	20
1	65	69	70	64	43
2	85	82	82	73	61
4	95	95	94	87	72
8	100	100	100	91	86
16	90	100	100	90	80
32	75	100	100	89	75

(b) SHA-256

Fault weight \ Number of faults	32	40	48	56
8	28	20	8	0
12	32	21	8	2
16	69	60	28	9
20	90	75	31	10
24	100	95	72	20
28	95	71	70	34
32	71	82	100	48

As described in Section 3.5.4, we inject faults in four different rounds and collect information about the correct and faulty hashes. We experimented with an equal number of faults in each of those four rounds. As listed in Table 3.1b, we were able to recover the target bits using 32 faults in the 24-bit fault model. While Hao et al. [HLMS14] use 65 faults in a 32-bit random fault model, our method is able to finish the search with 48 faults in the same fault model. These two data points are highlighted in Table 3.1b.

3.6.3 Performance of the Solver

Here we discuss the performance of our programmatic AFA solver on solving SHA fault instances. In Figures 3.4 and 3.5 you can see the cactus plot of MapleSAT solver and the extended versions of MapleSAT with the programmatic interface. We have turned each of the programmatic callbacks on and off to see which of them contributes more to the performance of the solver. There are four solvers compared in the plot. The base version of MapleSAT, MapleSAT with the SHA propagator, MapleSAT with SHA conflict analyzer and MapleSAT with both of these callbacks. We also experimented with Opturion CPX [Opt], which is a constraint solver that combines CP and SAT solving techniques, and won several medals in Minizinc challenge 2015. But unfortunately, it performed very poorly on our benchmark and could solve only a few instances of 32-bit fault model. The timings in the plot belong to the 32-bit fault model with 11 faults injected in SHA-1, and 48 faults

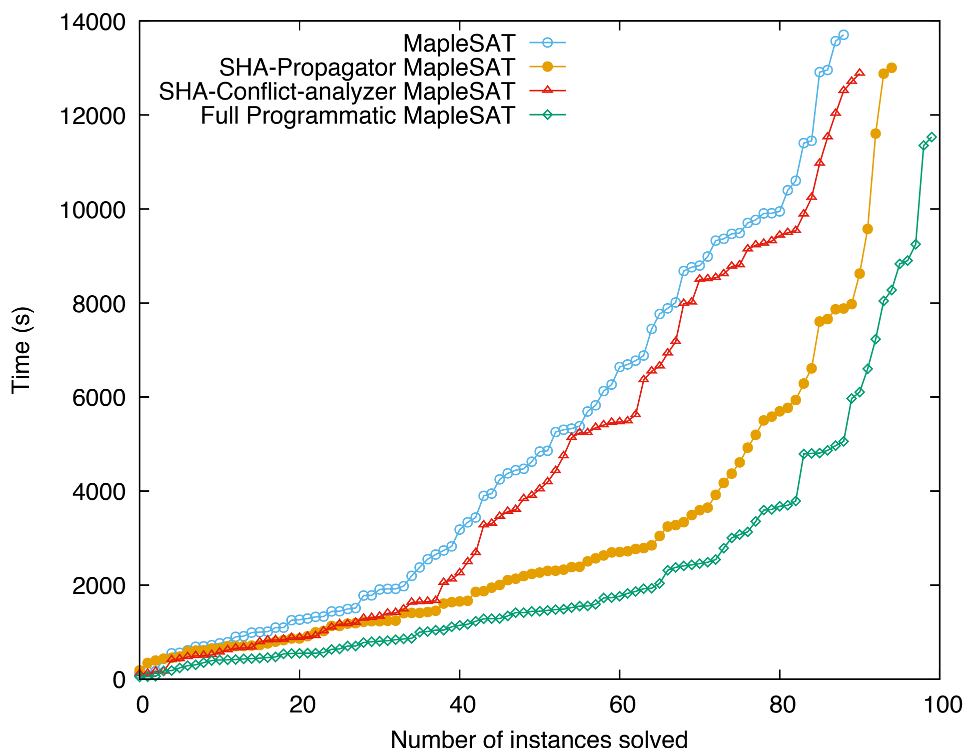


Figure 3.4: Comparison of MapleSAT and its programmatic versions on 32-bit fault model AFA on SHA-1.

in SHA-2. The plot shows that by embedding the external verification loop inside the SHA conflict analyzer and early detection of inconsistent solutions (rather than waiting for the instance to be completely solved), we can solve two more instances in SHA-1 and 14 more instances in SHA-2. But the main performance boost belongs to the propagation enhancement, in which the solver solves 6 more instances within the time limit in SHA-1, and 28 more instances in SHA-2. From the point of view of the number of faults, the lowest number of faults that base version of MapleSAT can recover the secret bits for all of the random messages is 14, wherewith the programmatic MapleSAT, it is 11. For the case of SHA-2, the gap is larger and the base version needs at least 64 faults, versus 48 faults needed by programmatic MapleSAT. Comparing the total timings for solving all of the instances in a fault model, between MapleSAT and programmatic MapleSAT, if we set the runtime of timed out instances to the time limit, we can see a 2.48x speedup in SHA-1 and 7.73x speedup in SHA-256. If we use the PAR-2 method (penalizing the timed out instances by setting their runtime to double the time limit), which was used in SAT

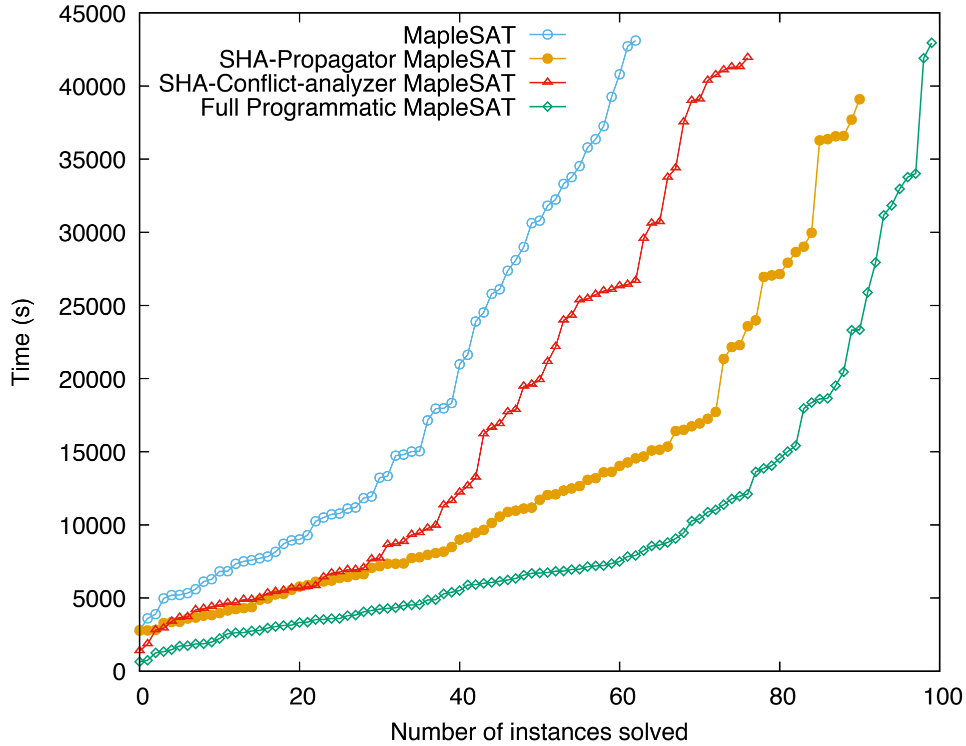


Figure 3.5: Comparison of MapleSAT and its programmatic versions on 32-bit fault model AFA on SHA-256.

competition 2017, we see a 3.16x speedup in SHA-1 and 14.3x speedup in SHA-2.

3.6.4 Discussion

Our results show the versatility of programmatic SAT solver architecture. The key insight is that by taking a state-of-the-art general purpose SAT solver and tailoring it to our cryptographic problem, we achieved considerable performance improvement. Looking at Table 3.1, one can observe that the data in certain rows suggests that when more faults are injected, fewer instances are solved. At first, it might seem counter-intuitive because adding more faults helps restrict the search space and hence should improve solver performance. However, note also that with every added fault equation, the number of clauses in the input to the solver grows rapidly (especially in the case of SHA-256), which can crucially slow down propagation. Thus there is a trade-off between search space reduction and

formula size that the cryptanalyst has to contend with. In our work we have limited the hash functions to a one-block version of Merkle-Damaård, i.e., we assume that the input message fits into the block size. Therefore the chaining value is set to the initialization vector value. One of the most practical applications of fault attacks on hash functions is the key recovery of HMACs. In that case, the input size is more than one block, and the internal chaining values dependent on the secret key and the message, which are given to generate a HMAC tag. Fortunately, our framework can be easily adapted to this scenario. Targeting the last block of the HMAC, an almost universal forgery attack as explained in [HLMS14] can be mounted on the HMAC-SHA-1 or HMAC-SHA-256.

AFA framework, in general, is much more flexible than DFA and it can usually be mounted with fewer faults in the same fault model, but comparing to the DFA it can be seen that the simulation time is significantly higher. That is because DFA equations are crafted by cryptanalysts that are specially designed to make the search space very small, and on the other hand, AFA equations are more generic. Therefore a longer search time is required. It is clear that the success of AFA partly relies on the power of the search tool it is using. Thus a more powerful (and maybe dedicated) search tool is desirable.

In this work, we tackled the arc-consistency problem of encoding of multi-operand adders by strengthening the propagation routine via a programmatic approach. However, the other method for coping with the arc-consistency problem is to use a different encoding that is arc-consistent. We have encoded our instances of AFA on both SHA-1 and SHA-256 using reduction of multi-operand addition to a series of two-operand additions and then using ripple carry encoding for each two-operand addition. These encodings were all I/O arc-consistent (according to definition 3.4.1). The instances were significantly larger than the base Nossum encoding and our solver had timed out on all of the generated instances with this method.

3.7 Differential Cryptanalysis

A naive way of encoding an algebraic collision attack is to have two copies of a function f that have constraints for having the same output and different inputs ($f(x) = f(y) \wedge x \neq y$). To improve upon this encoding, we can add a set of difference variables for all of the input, output, and intermediate variables in the two copies, where each difference variable is the XOR of the two corresponding variables in the two copies. These difference variables are building the differential characteristics (or path).

To express the set of possible combinations of a pair of bits x and x', the generalized

conditions [DCR06] is commonly used. It allows us to describe and encode the propagation of information through a differential characteristic. This notation is listed in the table 3.2.

Table 3.2: Notation for all generalized conditions. Each character represents the set of possible values for a pair of bits.

(x_i, x'_i)	?	-	x	0	u	n	1	#	3	5	7	A	B	C	D	E
(0, 0)	+	+		+					+	+	+		+		+	
(1, 0)	+		+		+				+		+	+	+			+
(0, 1)	+		+			+				+	+			+	+	+
(1, 1)	+	+					+					+	+	+	+	+

Just having the differential characteristics does not necessarily make the problem easier. However, by selecting a sparse differential path that is highly probable, the allowed combinations for variables in the two copies will reduce drastically. Note that for any operation, when we have - (no difference) in the input variables, we will have - at the output variables as well, i.e. running a function on the same input twice results in the same output. A sparse differential path means that most of the difference variables are forced to be -, and there should be few "difference"s (x), to ensure different inputs and keep the possible combinations throughout the differential path limited. We put "unknown" (?) in the places that the effect of having difference in earlier steps can potentially be canceled (to be found by the solver). The common approach to differential cryptanalysis of hash functions is to find a differential path first (starting from a sparse path, find the values for ?s), then use these constraints to find a conforming pair of messages that go through the two copies of the function that we had. It is possible that there are no pairs of messages that follow the path. In that case, we have to go back to the path and modify it. An important step in this process is the propagation of information throughout the differential path. In other words, having difference in the input of smaller operations, what is the possible set of combinations at the output of those operations (output difference). The implication from input differentials to output differentials is referred to as *propagation rules*.

Mendel et al. [MNS13] developed a dedicated tool for differential cryptanalysis of SHA-256. Prokop [Pro16] took their work and encoded their differential tables into SAT and studied the performance of different SAT solvers on them. Prokop shows collisions on SHA-256 up to 24 rounds, which is not matching the performance of Mendel's solver that gives a collision up to 31 rounds in the same attack model. Prokop is using bitwise XOR differences for encoding the difference possibilities. This means that he is using only ? (unassigned), - and x values for a difference variable. The advantage of this approach is that each difference variable can be encoded with a single Boolean variable. But the

disadvantage is that the propagation of information is less concrete in many cases. That is because a condition of for example `A` can not be expressed and thus it needs to fall back to the under-specified condition of `?`. To address this problem one can use multiple Boolean variables to encode each of the difference variables to cover all the possible information that is being propagated. The advantage of this approach is having more concrete possibilities and a more constrained set of values for pairs of message bits, but the disadvantage is that the instance becomes very large in terms of variables and clauses and the gain of having differential path constraints will be overshadowed by the complexity of the encoding. This is an opportunity for a programmatic component to implement the multi-valued logic of generalized conditions for difference variables while keeping the encoding of differential path simple. For example when using single Boolean variables, we can derive 2 propagation rules for the Boolean function $\text{IF}(x, y, z) = (x \wedge y) \vee (\neg x \wedge z)$, that are "`---` $\rightarrow$ `-`" and "`-xx` $\rightarrow$ `x`", and for the rest of input difference combinations, we can not imply any differential information for the output. But considering a multi-valued logic, we can have very fine-grained rules that rule out certain combinations for the pairs of bits at the output. Enumerating all of them gives us 1846 rules, which is expensive to encode in CNF.

In our implementation of programmatic propagation, simply put, we provide a truth table for each operation, that given input differences, determines and enforces the output difference if it is not `?`. Programmatic conflict analysis checks if the implied set of combinations of a difference variable does not have an intersection with a currently decided/deduced combination set. In other words, it looks whether after applying a propagation rule the difference variable becomes `#`.

We took the differential path starting points from Prokop [Pro16], but used our own encoding to translate the SHA-256 relations and differential path information into SAT. For encoding multi-operand addition we used Nossum's encoding [Nos12]. We ran MapleSAT (with and without the programmatic components) on these instances with a 24-hour time limit on Intel Core i7 CPU @ 3.4GHz and 16 GB of RAM. In the table 3.3, MapleSAT(Crypto) refers to the version of MapleSAT that we instrumented with programmatic callbacks. As timings show, not only we can increase the number of rounds from 24 to 25, but also we can solve the instances of 25 rounds roughly 2.3 times faster when we use the programmatic interface.

3.8 Related Work

Early works on the use of SAT solvers for cryptanalysis like finding cryptographic keys [Mas99], modular root finding [FMM03], or collision attack on MD5 [MZ06], only used

Table 3.3: CPU times (in seconds) for SAT-based differential cryptanalysis (finding collisions) in 25 rounds of SHA-256.

Solver	Encoding	Runtime (s)
MapleSAT	Prokop [Pro16]	29771.80
MapleSAT	Our encoding	21926.60
MapleSAT(Crypto)	Our encoding	12532.32

direct encoding of their problem to employ the power of SAT solvers. Subsequent works studied different ways of encoding the same problems into SAT to find formulas that are easier for a SAT solver in practice. Nossum [Nos12] and Morawiecki et al. [MS13] presented instance generators for preimage attack on SHA-1 and SHA-3. To make the SAT-based attacks more powerful, De et al. [DKV07] made use of Dobbertin's attack. They encoded the additional constraint alongside the main function to improve the base preimage attack on MD4. These types of cryptanalytic techniques can be encoded inside cryptographic reasoning components of the CDCL(Crypto) to keep the size of instance small, but still have the benefit of reducing the size of search space.

The problem of finding the highest probable linear/differential trail has been studied for lightweight ciphers like Simon [KLT15] and Speck [ADWL17]. In these works, the task of finding an optimal trail is defined as an optimization problem, and at each step, an SMT solver (in particular STP [GD07]) is queried with a trail and a parameter. If the solver returns SAT the parameter is increased and the process is repeated until the optimal value is reached.

Not all of the SAT-based cryptanalysis works have been completely blackbox. There were limited attempts to change the heuristics of the solver to improve the runtime. For example, Semenov et al. [SZBP11] changed the default activities and decay factor of VSIDS branching heuristics of Minisat and got better results. Although it should be mentioned that one can see this approach as configuring the parameters of the solver and not changing the algorithm. Prokop [Pro16] changed the branching heuristic of Lingeling to focus on the differential variables first in differential cryptanalysis of SHA-256. Furthermore, he studied value selection heuristics. For improving runtime of preimage attack on SHA-1 instances, an adaptive restart policy [NLG+17] and a splitting heuristics for divide-and-conquer parallel SAT solvers [NNS+17] has been proposed.

Notable SAT-based tools that have been developed specifically for cryptanalysis (at least initially), include CryptLogVer [MS13] and Transalg [OSG+16] which are tools for encoding cryptographic functions into SAT, CryptoMiniSat [SNC09] which includes XOR reasoning, and CryptoSAT [LNJVH14] and CryptoSMT [Ste] that provide higher level

languages for expressing cryptographic relations. For solving the algebraic equation set of the cryptosystem, SAT and SMT solvers are usually used. But other types of solvers have also been shown beneficial. Mouha et al. [MWGP11] use Mixed Integer Linear Programming solvers to find security boundaries in block ciphers.

Other than using off-the-shelf solvers, researchers have developed dedicated solvers to attack cryptographic primitives. Mostly these dedicated tools are based on guess-and-determine approach [Bar09], which is a method in algebraic cryptanalysis. In this method, we pick one variable with unknown value, guess a probable value for it, and then propagate the guessed information through the algebraic equation set that represents the cryptographic function, and in case of conflicting information, undo the guesses until the conflict is resolved. This is very similar to the process that a CDCL SAT solver follows (decision followed by unit propagation, and backtracking), but can be implemented specific to the function and not necessarily be in Boolean level. Mendel et al. [MNS11] developed a tool for differential cryptanalysis of SHA-256. They used random branching, problem specific propagation, and backtracking. They improved their results by improving the search strategy, better local collisions, and extra constraints [MNS13]. Eichlseder et al. [EMS14] took it further and improved the tool for SHA-512, by studying different branching heuristics. Although this tool is dedicated to this particular problem, it borrows many ideas from SAT solving. However, it is missing one of the most powerful components of a CDCL solver, which is conflict analysis. CDCL(Crypto) has the potential to implement the higher level logic on the propagation of information, and at the same time, use the underlying conflict analysis of the core CDCL solver on the Boolean level representation of the relations.

The research on fault attacks on SHA-like cryptographic structures was started by Li et al. [LLG09], where they applied a DFA on SHACAL-1, a block cipher based on the structure of SHA-1. Hemme et al. [HH11] extended their attack to SHA-1. The challenge of applying DFA on SHA-1 is the following: after applying the compression function of SHA-1 on the initialization vector (IV) and the message words, the value of IV is added to the output to make the chaining value for the next block. Hemme et al. handled this addition layer with separate fault injections and then launched an attack similar to [LLG09]. This is a key reason why their attack needs more than a thousand faults to be applicable. Our results show that an AFA can succeed with a far fewer number of injected faults. In the same fault model of 32 bits, we can find the secret bits with 11 faults.

Jeong et al. [JLSH13] proposed a fault attack on the HMAC setting of SHA-2 and showed that key values of size n can be recovered with approximately $n/3$ faults. Hao et al. [HLMS14] presented an AFA on SHA-2. They first perform a round of fault injections to recover the last internal state before the final addition. Then they inject some more

faults, encode and solve 4 rounds of SHA-256 at a time, fixing the found values at each step for the next solving step. This approach keeps the size of each fault instance small, but the problem is that if the found solution is inconsistent with the chaining input and correct hash value in the final solution, there is no comeback and no fixing mechanism is used. They use 65 faults in total for recovering the last 16 message words and hence full state recovery of SHA-256. They use STP [GD07] for solving the algebraic equations. For the same fault model, we can recover the secret bits with far fewer faults than their work. In the 32-bit fault model, we achieve the same results with 48 faults.

3.9 Chapter Summary

We presented a framework for SAT-based cryptanalysis inspired by the CDCL(T) paradigm. CDCL(Crypto) consists of a core Boolean SAT solver that is instrumented with programmatic callbacks for propagation and conflict analysis. These callbacks will contain user-provided cryptographic reasoning, similar to a T-solver in CDCL(T). This framework helps to have the higher level semantics of the cryptographic primitive available while keeping the size of the encoded function into SAT small and practical for the core SAT solver. CDCL(Crypto) enables the researchers to implement their cryptanalytic techniques on top of a powerful search engine. This framework has been applied to algebraic fault analysis of SHA cryptographic hash functions and resulted in much more effective search that requires far fewer number of injected faults compared to the previous best fault attack methods aimed at SHA-1 and SHA-256. Our programmatic solver (MapleSAT solver with SHA-enhanced conflict clause analysis and propagation) can achieve a speedup of up to 14x compared to the baseline solver. Also, a work in progress on the application of this framework on differential cryptanalysis has been demonstrated in this paper, which improves the number of rounds and the runtime of finding a collision for a round-reduced version of SHA-256 with 25 rounds. Symmetric cryptographic function designers usually test their designs against known attacks and cryptanalysis techniques. Automating these techniques helps with speeding up the design cycle. We believe that this framework has a great potential for improving the blackbox SAT-based cryptanalysis and therefore a valuable step toward automating cryptanalysis of cryptographic primitives.

MODIFIED BAYES' THEOREM:

$$P(H|X) = P(H) \times \left(1 + P(C) \times \left(\frac{P(X|H)}{P(X)} - 1\right)\right)$$

H: HYPOTHESIS
X: OBSERVATION
P(H): PRIOR PROBABILITY THAT H IS TRUE
P(X): PRIOR PROBABILITY OF OBSERVING X
P(C): PROBABILITY THAT YOU'RE USING BAYESIAN STATISTICS CORRECTLY

XKCD (https://xkcd.com/2059/)

Chapter 4

Initialization of SAT Heuristics

In recent years, it has been shown that machine learning (ML) based heuristics for branching and restarts can dramatically improve the performance of SAT solvers [LGPC16a, LVP+17, LOM+18]. This impact has been forcefully demonstrated by the success of the MapleSAT solver and its variants (e.g., MapleCOMSPS) [LGPC16a, LVP+17, LOM+18] in winning medals at the highly-competitive SAT competition in 2016 and 2017 [BH+16, HJB17]. Many solvers that have won medals in subsequent years also use the ML-based branching and/or restart heuristics developed first in MapleSAT [LGPC16a].

This impact can best be explained via the view that solvers are fundamentally proof systems, and machine learning methods are powerful ways of initializing, sequencing and selecting proof rules to optimally and adaptively solve formulas. Inspired by the success of machine learning in the context of the MapleSAT solver (and variants), in this work we propose a set of online Bayesian Moment Matching (BMM) based methods to solve the initialization problem in SAT solvers.

The Initialization Problem in SAT Solvers. We define the initialization problem as follows: given a SAT formula ϕ, compute an *initial order* over the variables of ϕ and values/polarity for them such that the runtime of CDCL solvers on input ϕ is minimized. By initial order, we mean a total order over variables chosen by the CDCL solver S (and similarly, by initial value assignment we mean a mapping from variables to truth values) at the beginning of its search, i.e., before any variables have been branched upon by the solver S. Solver developers have known for a long time that the *initial order and value assignment to the variables of an input formula* can have a significant impact on the performance of CDCL SAT solvers.

BMM-based Method to Solve the Initialization Problem in SAT Solvers. The BMM method proposed in this chapter is used as a pre-processor to a CDCL SAT solver. Our method takes as input a SAT formula ϕ and outputs a total order and assignment over the variables of ϕ. The method assigns a Bernoulli random variable to each variable of the input formula ϕ, associated with an unknown probability p of the variable being set to true (and $1-p$ represents its probability of being false). For every clause C in the input formula ϕ, the belief about p is updated using Bayesian inference and moment matching. After our BMM method has scanned all the input clauses, it arrives at a posterior distribution that suggests an assignment that *ideally* satisfies most of the clauses (if not all of them).

The posterior distribution thus obtained is used to construct an assignment A that is most likely to satisfy the formula ϕ. One could treat such an assignment as a good guess for a satisfiable assignment to the formula ϕ (assuming it is satisfiable). Even if the formula is unsatisfiable, the hypothesis of our work is that the assignment A can be used as a good initial value (aka, polarity) selection for the variables in ϕ, as the CDCL solver starts its search. Further, the variables can be ranked in decreasing order based on the probability associated with their truth value in A (more certain the BMM is about a variable's value, the higher it is in the variable selection ranking). This ranking can be used as an initial variable selection order by the CDCL SAT solver's branching heuristic.

An additional important point about our approach is that when the clause-learning method in the BMM-enhanced CDCL solver deductively learns a unit or a binary clause, it is used to update the posterior probability of the variables appropriately. The motivation behind this corrective feedback method from clause learning to the posterior probabilities of variables is that these BMM-based polarities are used to guide the solver's polarity/value selection heuristic during the run of the solver (not merely during the initialization), and thus get a further boost in performance.

We perform extensive experiments to test the efficacy of our BMM-based heuristics against state-of-the-art solvers. We show that BMM-based initialization of variable order and value selection in the context of CDCL SAT solvers can be effective for real-world instances obtained from verification, program analysis, software engineering and crypt-analysis.

4.1 Contributions

1. **BMM-based Initialization Method.** We present the design and implementation of a novel BMM-based initialization method to address the "initialization problem"

for value selection and variable order in CDCL SAT solvers. The key idea is to use clauses in the input formula as evidence to update a probability distribution of value assignment for each variable in the input formula. Our method can incrementally update and improve the posterior probability during the search by taking into account unit and binary learnt clauses in a corrective feedback loop. (Section 4.2)

2. **Evaluation on Cryptographic Instances.** We perform an apple-to-apple comparison of BMM-based versions of CryptoMiniSAT, MapleSAT, and Glucose against their respective configurations using 4 other initialization methods on a set of hard cryptographic benchmarks encoding round reduced SHA-1 inversion attacks, with a timeout of 4 hours. We used these solvers since they are among the best solvers for hard cryptographic instances. More precisely, for each solver, we compared our BMM-based method against 4 other initialization methods (namely, default, random, Jeroslow-Wang [JW90], and Survey-propagation [BMZ05]). Our BMM-based method significantly outperforms all other methods, where BMM-based MapleSAT inverts all of the given targets and BMM-based CryptoMiniSAT solves the instances 50% faster on average. (Sec. 4.4.1)

3. **Evaluation on SAT 2018 and 2019 Application Instances [HJS18a, HJS16].** We further compare the efficacy of BMM-based versions of MapleLCMDistChronoBT (winner of SAT 2018 competition) and MapleCOMSPS (Gold/Silver medalist in SAT 2016/2017 competition), against the corresponding respective versions with 4 other initialization methods (listed above). We observe that our BMM-based method outperforms all other versions with 12 additional instances solved and an average runtime speedup of 15.2%, compared to the next best method, namely, Jeroslow-Wang. (Section 4.4.2)

4.2 Bayesian Moment Matching as a SAT Solver Component

We adopt the algorithm designed by Poupart, Jaini and Duan described in 2.5.2, for the formulation of finding a solution to a satisfiable instance of Boolean SAT problem in a BMM setting. We show how the posterior distribution learned by BMM can help solve the initialization problem in SAT solvers.

The learned probabilities collectively represent an assignment to the variables that maximizes the number of satisfied clauses. For a relatively small Boolean formula, the

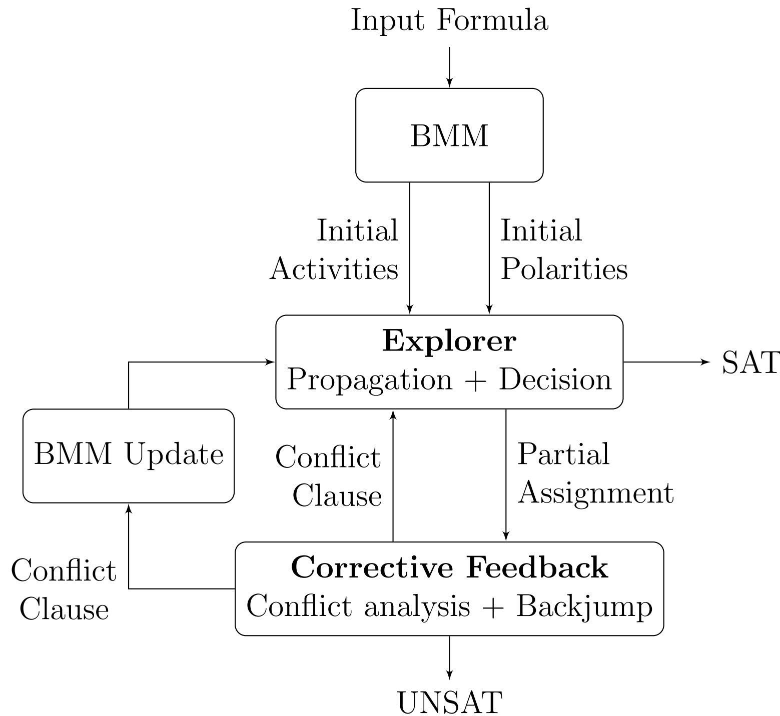


Figure 4.1: Overview of BMM as a component in a SAT Solver.

BMM algorithm can converge to a solution, if it has one. As the problem size grows, the chance of approximating a correct assignment decreases, and the computation time increases. Thus we might not be able to use this method as a standalone SAT solver for large scale problems with millions of variables and clauses. However, as we show, the learned probabilities are very valuable for a CDCL search to arrive at a solution significantly faster than starting at random or using other initialization methods.

Modern CDCL SAT solvers commonly use *look-back* heuristics, which means that they collect statistics about their variables and clauses during the search, and maintain scores for variables so that they can make educated guesses in the future. For example, we know that CDCL solvers (e.g., VSIDS or LRB branching heuristics) tend to pick branches that are more likely to prove a subspace unsatisfiable faster than methods that don't maintain such statistics [LGZ+15]. At each decision step, SAT solvers ask two questions: which unassigned variable to pick (Branching/Variable order heuristics) and what value to assign to that variable (Polarity/Value selection heuristics). It is well known that these heuristics have a huge impact on the performance of a SAT solver. An important question

as described in the introduction is how to initialize the variable values and the variable selection order at the start of the search where there are no previously seen data. This question is generally referred to as the *initialization problem.*

BMM-based Initial Value Selection. As the learned probabilities collectively represent an assignment to the variables of the input formula that satisfies most of the clauses, it is natural to hypothesize that they can be used as initial preferred values. We use BMM as a pre-processor that scans the clauses and computes the preferred initialization values before the search starts. We use 10 epochs for application instances (each clause is seen ten times), and empirically observed that it results in a good initial point while being efficient. We simply set the preferred value of a variable to True if the first moment $\mathbb{E}(\theta_x) > 0.5$ and False otherwise.

BMM-based Initial Variable Selection. Successful branching heuristics like VSIDS [MMZ+01] and LRB [LGPC16a] that are widely used in modern SAT solvers, keep a score for each variable, called activity, which represents how much that variable was involved in conflict analysis recently. The variable with the highest score will be picked as the decision variable. At the start of the search, we do not have any information about the variables and which one is preferred over the others. Therefore it is very common to start from zero scores for all variables and build the ranking of variables based on the search statistics. However, having the learned probabilities, we can prioritize the variables before the search starts. In our experiments, we give higher priority to the variables with less uncertainty about a polarity (high probability of being either True or False). For each variable x, we define the $score(x)$ to be a number in the range [0.5, 1] as follows:

$$score(x) = \begin{cases} 1 - \mathbb{E}(\theta_x), & \mathbb{E}(\theta_x) < 0.5 \\ \mathbb{E}(\theta_x), & \mathbb{E}(\theta_x) \geq 0.5 \end{cases}$$

This is the same as saying for a variable x with the $Beta(\alpha_x, \beta_x)$ distribution: $score(x) = \max(\alpha_x, \beta_x)/(\alpha_x + \beta_x)$. The score will be 1, if BMM is certain that the variable x is False ($\mathbb{E}(\theta_x) = 0$), or True ($\mathbb{E}(\theta_x) = 1$), in a satisfying assignment.

Updating Posterior During Search. During a CDCL search, the solver might reach a conflicting state, where the partial assignment to the variables cannot be extended to a full assignment. At that point, the solver analyzes the root cause of this conflict and encodes this information as a clause (*conflict clause*). Conflict clauses are implied by the

original formula, so they can be added to the original formula. The conflict clauses can thus be treated as new evidence. In the case that we use BMM probabilities to initialize polarities, the partial assignment that led to a conflict is derived from the BMM posterior distribution. This means that the new evidence has the necessary information to fix an inaccurate posterior. We update the posterior using this corrective feedback. However, we do this only for unit and binary clauses to keep the overhead low. We directly update the polarity of variables in the conflict clause.

Figure 4.1 shows a high level block diagram of where BMM fits in as a component in a CDCL SAT solver. The "Propagation + Decision" block is responsible for expanding the partial assignment by assigning values to unassigned variables and propagating this information to other variables. This block receives initial values for the order of variables and their preferred values from BMM. The "Conflict analysis + Backjump" is responsible for correcting the mistakes made by the explorer component. The BMM Update unit gets a copy of the conflict clause returned by this component and updates the probabilities. In other words, an approximate solution proposed by BMM is checked on the formula (by propagation), and if it does not satisfy the formula, the conflict analysis component gives corrective feedback about the inaccuracy of the probabilities.

4.3 Description of Other Initialization Methods

Default. Most CDCL solvers simply initialize the activity scores of variables with zeroes and set the preferred polarity of variables to false. In this work whenever we say default or do not explicitly mention the initialization method, we mean the all zero and all false initialization.

Random. To verify that our proposed initialization method indeed improves the search and not randomly shuffles the variables and values, we compare with random initialization as a control experiment. In this method, polarities are randomly picked with 0.5 probability between true and false, and activity scores are set to a number between 0 and 1 picked uniformly at random.

Survey Propagation. Survey propagation is a message passing algorithm that was designed to find solutions for random k-SAT problems [BMZ05]. They are mostly believed to be the hardest to solve when their clause to variable ratio is close to the experimental threshold of SAT-UNSAT regions. Survey propagation works over the factor graph

representation of SAT instances. It generates messages that survey over clusters of ordinary messages and then uses these surveys to fix variables and simplify the problem by decimation.

Jeroslow-Wang. Jeroslow and Wang proposed a static branching heuristic [JW90], which in some modern solvers is used for computing initial scores for literals. It assigns a score to each variable such that the variables that appear in shorter clauses get a higher score. The intuition is that these variables when assigned by the solver, create unit clauses sooner than others, and allow unit propagation to imply many other literals. The score for each variable is computed as $score(x) = \sum_{x \in C, C \in \phi} 2^{-|C|}$, where ϕ is the input formula and C is a clause in ϕ.

4.4 Experimental Results

In this section, we present and discuss the experimental evaluation of our BMM-based initialization method and compare it against 4 other initialization methods described in Section 4.3. We implemented the initialization methods in all of the solvers in a modular way. In other words, we kept all the implementation of solvers intact except for the initialization methods, so we can have an apple-to-apple comparison, between different versions of one solver.

For each combination of (solver, initialization method), we experimented with 3 configurations: initializing 1) polarities only, 2) activities only, and 3) both of polarity and activity. For each combination, we report the best performing configuration. We observed that generally on SAT 2018 and Cryptographic benchmark the third configuration performs the best, except survey propagation, where initializing polarity only performs better than the other two configurations. On SAT 2019 benchmark the best performing configuration was different for each initialization method, which is reported in Section 4.4.2.

4.4.1 Evaluation over Hard Cryptographic Instances

Experimental Setup. All jobs were run on Intel Xeon E5-2667 CPUs at 3.20GHz and 8GB of RAM. We used cryptographic instances encoding preimage of round reduced SHA-1 hash function. We encoded 22 rounds of SHA-1 and used 50 randomly generated hash values to be inverted. Time and memory limit for cryptographic instances was 4 hours and 8GB respectively.

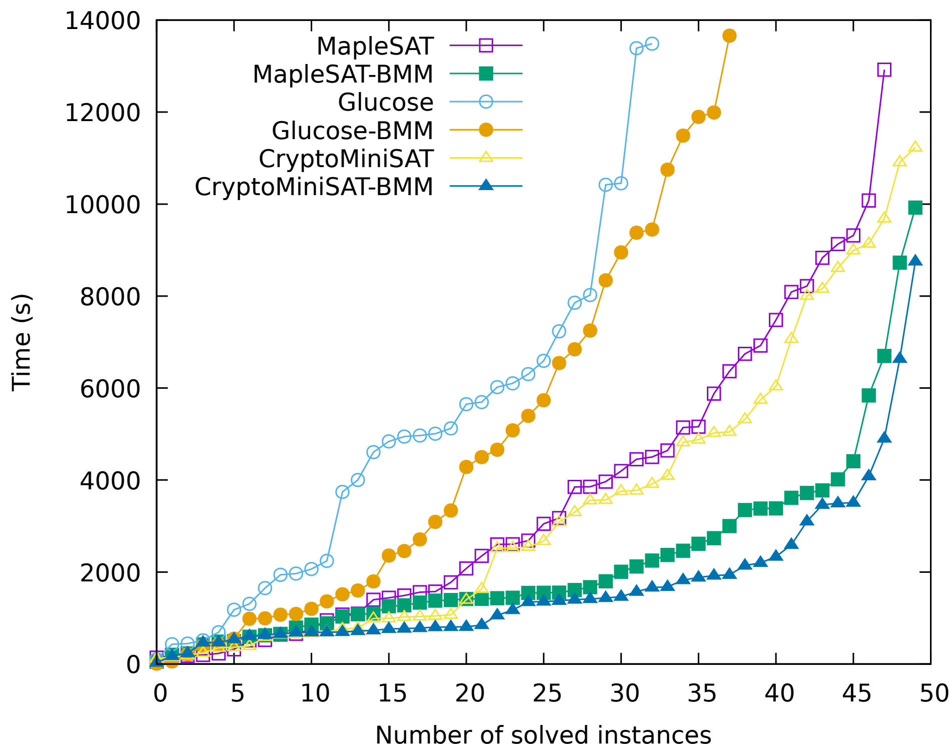


Figure 4.2: Performance comparison of MapleSAT, Glucose and CryptoMiniSAT solvers with default, and BMM initialization methods on hard cryptographic benchmarks.

Solver Descriptions. The solvers we used were MapleSAT [LGPC16a], Glucose-4 [AS18] and CryptoMiniSAT-5 [Soo18]. From the experiments performed on SHA-1 instances in the literature [Nos12, NLG+17, NNS+17], we know that these solvers are top performing solvers in this benchmark. We used 100 epochs for pre-processing and 1 epoch for updating BMM posterior.

Results. Table 4.1 gives details on the number of solved instances out the 50 hard cryptographic instances, where it can be seen that BMM version of MapleSAT is the only variant of MapleSAT that can solve all of the instances with much lower average runtime compared to other initialization methods. Also, BMM version of CryptoMiniSAT solves the instances around 50% faster than the default version on average. Figure 4.2 shows how MapleSAT-BMM and CryptoMiniSAT-BMM have a clear advantage over other versions of these three solvers.

Table 4.1: Number of solved instances out of 50 hard cryptographic instances and average runtime (in seconds) of MapleSAT, Glucose and CryptoMiniSAT with different initialization methods.

	Initialization	Total	Avg. time
MapleSAT	Default	48	3645.08
	Random	48	3180.42
	Survey Propagation	40	3405.20
	Jeroslow-Wang	47	3389.27
	BMM	50	2238.85
Glucose	Default	33	4817.69
	Random	32	5741.74
	Survey Propagation	30	5386.74
	Jeroslow-Wang	32	6334.71
	BMM	38	4563.08
CryptoMiniSAT	Default	50	3475.06
	Random	50	3223.48
	Survey Propagation	41	3501.00
	Jeroslow-Wang	49	5387.20
	BMM	50	1706.63

4.4.2 Evaluation over SAT Competition 2018 and SAT Race 2019 Application Instances

Experimental Setup. All jobs were run on StarExec environment with Intel(R) Xeon(R) CPU E5 at 2.40GHz [SST14]. We used the main track of the SAT competition 2018, which contains 400 instances coming from a variety of real-world application domains, like verification, graph problems, scheduling, and combinatorics [HJS19]. The SAT race benchmark is partitioned into "new" and "old" subsets, marking newly submitted instances to the competition and re-used instances from the past competitions. We used the "new" subset of the instances containing 200 instances. The time limit for solving each instance was 5000 seconds (the same as SAT competitions) and the memory limit was 8GB.

Solver Descriptions. The solvers that we used to incorporate BMM were MapleCOMSPS (gold / silver medalist of SAT competition 2016 / 2017) [LOG+17] and MapleLCMDistChronoBT (winner of SAT competition 2018) [RN18]. We used 10 epochs to compute the posterior in the pre-processing phase and 1 epoch for each learned unary and

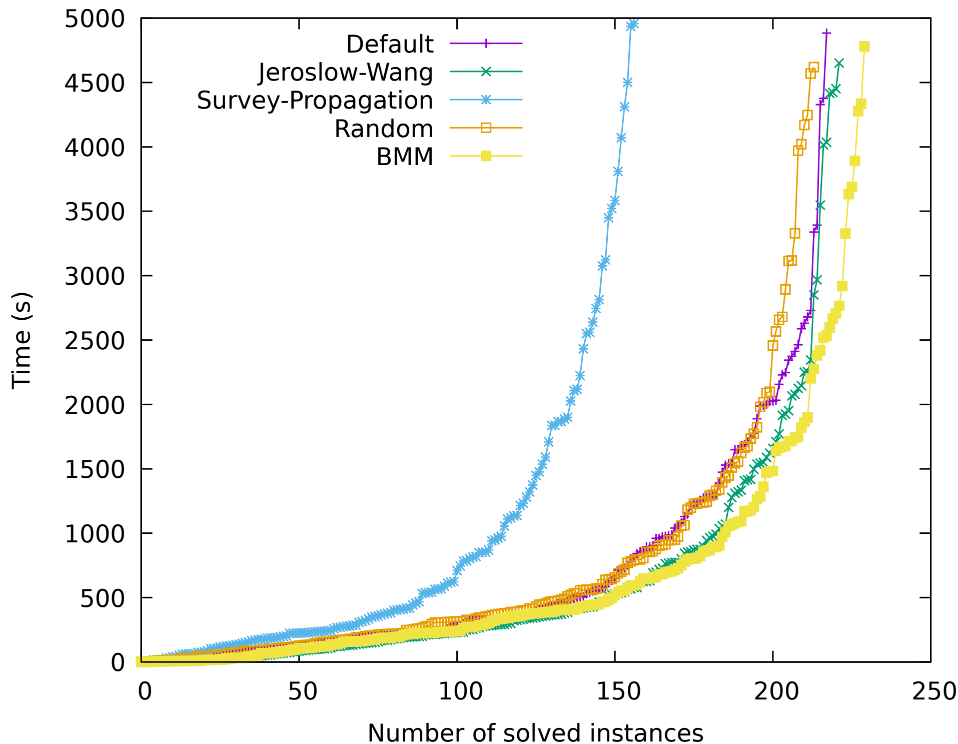


Figure 4.3: Performance comparison of different version of MapleCOMSPS on SAT competition 2018 benchmark.

binary clause. MapleLCMDistChronoBT, switches between Distance, VSIDS, and LRB branching heuristics. We initialized activity scores of all of these heuristics. Similarly, we initialized both VSIDS and LRB in MapleCOMSPS.

Results. Table 4.2 shows the number of solved instances out of 400 instances by the two solvers described above, comparing BMM with other methods. Figure 4.3 depicts that BMM-based initialization beats all other methods, by solving more instances, and having lower average runtime on the solved instances. The closest performing method is the Jeroslow-Wang, which solves 4 more than the default, but still, BMM solves 8 more instances than Jeroslow-Wang. In the case of MapleLCMDistChronoBT, BMM-based initialization does not improve the number instances, however, it solves the instances 15% faster on average.

Table 4.3 shows the number of solved instances out of 200 instances by the two solvers

Table 4.2: Number of solved instances (out of 400) and average runtime (in seconds) of MapleCOMSPS and MapleLCMDistChronoBT and their variations on SAT competition 2018 benchmark. SAT column shows how many of the solved instances were satisfiable.

	Initialization	Total	SAT	Avg. time
MapleCOMSPS	Default	218	124	674.43
	Random	214	121	678.09
	Survey Propagation	157	100	862.30
	Jeroslow-Wang	222	128	654.05
	BMM	230	136	646.18
MapleLCMDist	Default	240	138	769.85
	Random	232	131	673.02
	Survey Propagation	173	109	885.50
	Jeroslow-Wang	235	134	655.98
	BMM	240	139	652.80

described above, comparing BMM with other methods. Unlike the SAT 2018 benchmark, the best performing configuration was different among the initialization methods, which is listed in Table 4.3. For MapleCOMSPS BMM-polarity was the best configuration, and for MapleLCMDistChronoBT BMM-activity was the best performing configuration. In both of the solvers, BMM-based initializations are the best version of their respective solvers, beating the default version by 5 instances in MapleCOMSPS and 2 instances in MapleLCMDistChronoBT. It should be noted that BMM-based versions solve 5 more satisfiable instances compared to default MapleCOMSPS and 4 more satisfiable instances compared to default MapleLCMDistChronoBT.

4.4.3 Discussion of Experimental Results

SAT vs. UNSAT. The posterior distribution that BMM learns, is supposed to form a solution to the input formula. Therefore we expect to see better performance in satisfiable instances rather than unsatisfiable instances, and in fact that is what we have observed in our experiments. Tables 4.2 and 4.3 show that the BMM-initialized MapleCOMSPS, solves 12 more (5 more in 2019) satisfiable instances compared to the vanilla MapleCOMSPS, and solving the same number of unsatisfiable instances. All instances in our hard cryptographic benchmark are satisfiable (there exists a preimage to each hash target, and the task is to find it), and we specifically wanted to study this benchmark as an important class of satisfiable instances.

Table 4.3: Number of solved instances (out of 200) and average runtime (in seconds) of MapleCOMSPS and MapleLCMDistChronoBT and their variations on SAT race 2019 benchmark. SAT column shows how many of the solved instances were satisfiable.

	Initialization	Total	SAT	Avg. time	Best config.
MapleCOMSPS	Default	120	89	696.310	default
	Random	119	88	732.489	Activity-Polarity
	Survey Propagation	115	84	813.637	Polarity
	Jeroslow-Wang	123	92	712.904	Activity
	BMM	125	94	841.985	Polarity
MapleLCMDist	Default	120	88	604.368	default
	Random	119	89	685.499	Polarity
	Survey Propagation	115	83	946.500	Polarity
	Jeroslow-Wang	120	88	830.279	Activity-Polarity
	BMM	122	92	665.060	Activity

Impact of BMM Update. As described in Section 4.2, we also update the posterior with the new evidence (conflict clauses that are implied by the formula) that the solver generates. This update, had a positive impact on the performance, although not a significant impact. On average the solving times are reduced by 11.2% in application benchmark, but no additional instances were solved. The results in the tables and figures are with using the BMM update.

Sub-category Analysis for SAT 2018 Application Instances. We analyzed categories of problems in the SAT competition benchmark [HJS19], to further study which types of problems, BMM is more effective, and in which types it is less effective. The categories that we extracted from this benchmark were: Combinatorics, Cryptography, Graph theory, Verification, Number theory, Scheduling and Hard 3-SAT. In most categories, the BMM-based version of MapleCOMSPS performs on par with the default version. However, it solves one more instance in the verification category and one less instance in hard 3-SAT and scheduling problems, and a large leap of 16 more instances in the cryptography category.

Computational Overhead. In each epoch, all clauses are processed and for each clause, all of the literals in the clause are linearly processed, which means that the overall complexity is linearly proportional to the total number of literals appearing in the formula. 10

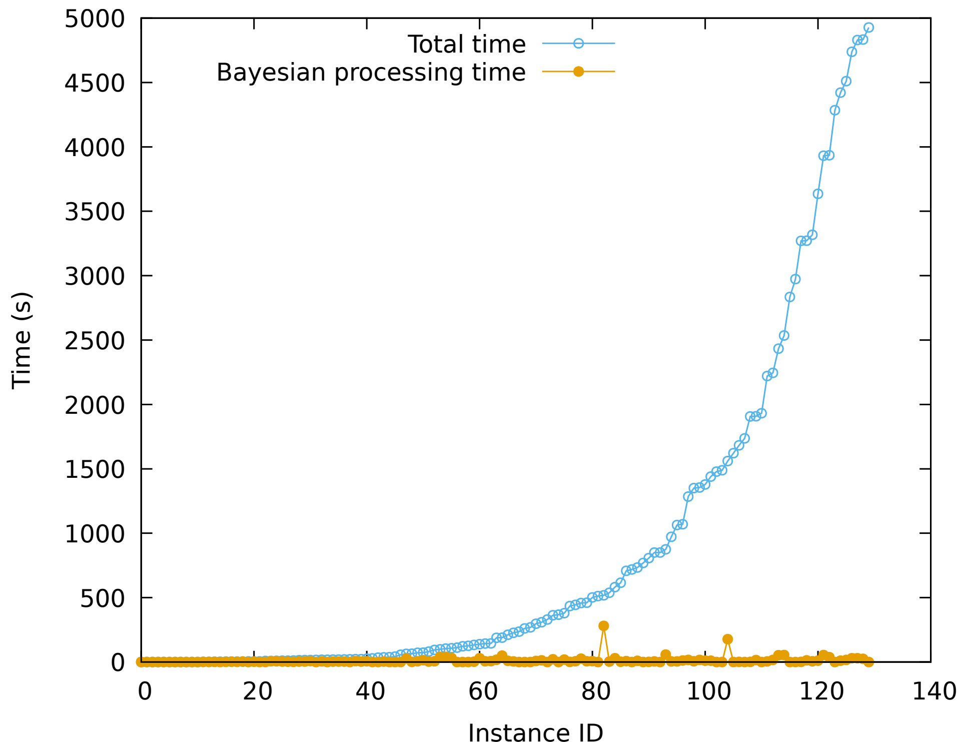


Figure 4.4: Comparison of total time a SAT solver took to solve an instance vs BMM preprocessing of the formula with 100 epochs on the SAT 2016 competition benchmark.

epochs over the largest formula in our benchmarks with 12 million clauses and 2 million variables, takes 80 seconds. On average BMM pre-processing constitutes 6% of the total running time of MapleCOMSPS on the SAT 2018 benchmark. This number is negligible in hard cryptographic instances even with 100 epochs.

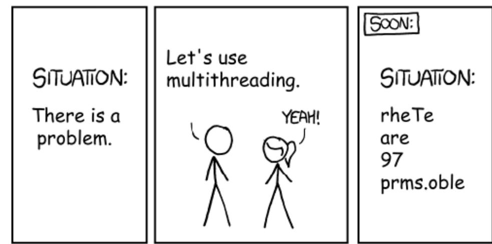
4.5 Related Work

Unfortunately, the initialization problem has not been studied as extensively as other components of the SAT solvers. Jeroslow-Wang [JW90] proposed a scoring system for each literal based on the length of the clauses that the literal appears in, where the literals that are appearing in shorter clauses are preferred. Initially, this was proposed as a static branching heuristic, but this was later also used as a way of giving initial preference to the literals. However, as the Boolean formula gets larger and more complicated, it might

not capture the information about the underlying structure. In contrast, BMM updates the prior hypothesis with the target of satisfying all of the clauses and does not use a proxy for guessing a good measure. Despite being an approximation, BMM takes us to a relatively useful starting point. Most of the modern solvers, set the polarities and activities either to a fixed value (all zero, all False or all True), or a random value and let the search engine explore the search space. Some solvers in their initial phase of exploring, use a different branching heuristic (e.g. Distance [XLL+19]) to get to a fruitful state and then use the main branching heuristics. However, all such solvers that use hybrid branching methods, only get to that desired state by collecting conflict clauses and do not re-use the intermediate activity scores. Kibria et al. [KL06], proposed a genetic programming approach to find initialization of activities that minimizes runtime, where they had mixed results on a small set of electronic design automation instances.

4.6 Chapter Summary

We used the design of a novel BMM-based algorithm for the initialization problem of value selection (polarity) and variable order (branching) heuristics in conflict-driven clause-learning SAT solvers. We implemented our methods alongside other initialization methods (random, survey propagation, Jeroslow-Wang and default) in state-of-art solvers such as MapleCOMSPS, MapleLCMDistChronoBT, MapleSAT, Glucose and CryptoMiniSAT, and showed significant improvement over these already leading solvers. We evaluated our methods on the main track benchmark of SAT competition 2018 and 2019, consisting of real-world application instances, as well as a set of hard cryptographic instances (inversion attacks) obtained from a round-reduced version of SHA-1 hash function. The BMM-enhanced version of MapleCOMSPS with both value selection and value order initialized, solves 12 more instances with lower average runtime compared to the baseline version, and is also faster than the random, survey propagation and Jeroslow-Wang initializations. Furthermore, the BMM-enhanced version of MapleSAT solves all of the hard cryptographic instances encoding preimage attacks on SHA-1 in our benchmark, and BMM-based CryptoMiniSAT solves them around 50% faster on average than the default version.



https://www.reddit.com/user/TheGreatCabbage2/

Chapter 5

Machine Learning based Parallel SAT

The availability of many-core machines has led to a considerable effort in parallel SAT solver research in recent years [BS18]. Broadly speaking, researchers have developed two parallel SAT solver strategies, namely, *portfolio* and *divide-and-conquer* (DC) solvers. A portfolio SAT solver consists of a set of sequential worker solvers, each implementing a different collection of heuristics, and all of them attempting to solve the same instance running on different cores of a many-core machine. The key principle behind a portfolio solver is that of *diversity of heuristics*, i.e., by leveraging a diverse set of heuristics to solve an instance one may be more efficient than just using a single heuristic given the well-known fact that different classes of formulas are often best solved by distinct methods [HJS08]. On the other hand, DC solvers partition the search space of the input formula and solve each sub-formula using a distinct sequential worker solver. Each sub-formula is a restriction of the input formula with a set of assumptions [ZBH96]. In both the portfolio and DC settings, the sequential worker solvers may share clauses to exchange useful information they learn about their respective search spaces.

In the context of DC solvers, a splitting heuristic is a method aimed at choosing the "next variable" to add to the current list of assumptions (also known as guiding paths [ZBH96]). A bit more formally, one can define a splitting heuristic as a function that takes as input features of a given formula ϕ and/or statistics of a DC solver's state and outputs a variable to split. Splitting heuristics are typically dynamic, i.e., they re-rank variables at regular intervals throughout the run of a DC solver. The process of splitting itself can be described as follows: for a given input formula ϕ, consider that a variable v is chosen for splitting. The solver generates $\phi[v = F]$ (resp. $\phi[v = T]$) by setting v to False (resp. True) and appropriately simplifying the resultant sub-formulas using Boolean constraint propagation. These two sub-formulas are then solved in parallel. Each of these

sub-formulas can be further split into smaller sub-formulas recursively. Many heuristics for splitting have been studied in the literature [HKWB11, ALST16, AHJP14, NNS+17].

Splitting heuristics can be broadly categorized as *look-ahead* and *look-back*. Look-ahead heuristics choose some subset of variables in the input formula, analyze the impact of splitting on these variables, and rank them based on some measure that correlates well with minimizing runtime[1] of the solver on the sub-formulas thus obtained. By contrast, look-back heuristics compute statistics on "how well a variable participated in the search exploration in the past" (e.g., in clause learning, propagation, etc.), rank them appropriately and split on the highest-ranked variable. Examples of look-back heuristics include VSIDS activity [ALST16], number of flips [LFBSK19].

While considerable work has been done on splitting heuristics, almost all previous approaches share the following characteristics: they compute some features of the input formula and/or statistics over the solver state at appropriate intervals during the solver's run, and then use these as input to a "hand-coded" function (a splitting heuristic designed by the solver designer), that in turn computes a metric correlated with solver runtime to pick the "best" variable to split. By metric we mean a quantity that can be used to rank variables of the input formula such that splitting on the highest-ranked variable ideally corresponds to minimizing solver runtime. We argue that the design of splitting heuristics can be dramatically improved by leveraging a data-driven machine learning (ML) approach, especially for families of formulas (e.g., cryptographic instances) where it can be hard for human designers to come up with effective "hand-coded" splitting heuristic.

In this chapter we focused on the splitting heuristic in divide-and-conquer parallel SAT solvers and present three look-back heuristics. We first present propagation-rate, an ad-hoc heuristic based on how much a variable impact other variables, when set. Next, we propose two ML-based methods, namely *pairwise ranking*, and *min-rank*. The pairwise ranking model takes as input features of a given formula ϕ, aspects of solver state, as well as features of a pair of variables v and u, and ranks them in descending order based on some splitting metric. This ML-based "comparator" is in turn used by our DC solver to rank variables for splitting at regular intervals during its run. The min-rank model, takes as input features of a given formula ϕ, aspects of solver state, and features of a variable v, and outputs whether the variable v has the minimum rank among all variables of the input formula (i.e is it the best variable to split?). Both of these models are binary classifiers implemented using *random forest*. We implemented our heuristics in the Painless parallel solver framework [LFBSK17] (we refer to our solver as MaplePainless), and compared it with top parallel SAT solvers from recent SAT competitions. We find that our ML-based

[1] Runtime is the wallclock time of solving a formula.

method out-performs the best DC solvers on both SAT 2018/2019 competition as well as cryptographic instances. We only compare our MaplePainless solver against the state-of-the-art DC solvers for the following reasons: first, it is well-known that DC solvers often outperform the most notable portfolio and sequential solvers on cryptographic instances. On the other hand, DC solvers are known to perform poorly relative to the portfolio and sequential solvers over application instances.

5.1 Contribution

1. **Propagation Rate Splitting Heuristic.** We present a new splitting heuristic based on the propagation rate, where a formula is broken into two smaller sub-formulas by setting the highest propagating variable to True and False. We evaluate the improved solver against the top parallel solvers from the SAT 2016 competition on the Application benchmark and a benchmark of cryptographic instances obtained from the encoding of preimage attacks on the SHA-1 cryptographic hash function. Our solver, called MapleAmpharos, outperforms the baseline AMPHAROS and is competitive against Glucose, parallel CryptoMiniSat5, Treengeling and Plingeling on the SAT 2016 Application benchmark. Additionally, MapleAmpharos has better solving time compared to all of the solvers on our crypto benchmark.

2. **MaplePainless: A DC Solver based on ML-based Splitting Heuristics.** We present MaplePainless, a DC solver that leverages ML for splitting. Briefly, our splitting heuristics are ML models, trained offline on both static formula/variable features (e.g., variable occurrence in binary clauses) as well as "dynamic" features based on aspects of the solver's state at runtime (e.g., number of times a variable has been assigned, activities). We propose and implement two different models, namely, *pairwise ranking* and *min-rank*, described above. At runtime, the trained ML-model is invoked by MaplePainless on a vector of static and dynamic variable features at appropriate intervals, which in turn outputs a ranking of the variables in the input formula. The splitting heuristic then chooses the top-ranked variable, splits the formula by assigning that variable both true and false values, and gives the resultant sub-formulas to worker solvers to solve.

3. **Evaluation on Cryptographic Instances.** We evaluated our splitting heuristics on a cryptographic benchmark encoding preimage attack on a round-reduced SHA-1 function (inversion of 60 random hash targets). We used top sequential solvers in solving cryptographic instances as backend solvers (MapleSAT and Glucose). We

outperform the baseline solver (Painless-DC with the same backends and Flip as splitting heuristics) in an apple-to-apple comparison, and also other top DC solver, Treengeling.

4. **Evaluation on SAT Application Instances from SAT 2018 competition and SAT 2019 race.** We evaluated our splitting heuristics on main track benchmarks of SAT competition 2018 and SAT race 2019 against the baseline solver (Painless-DC with Flip as splitting heuristic) in an apple-to-apple comparison, and also against Treengeling (state-of-the-art divide-and-conquer solver from recent SAT competitions). On the combined SAT 2018 and SAT race 2019 benchmark, we outperform the baseline solver as well as Treengeling both in terms of the number of solved instances and the PAR-2 score. Furthermore, MaplePainless solves satisfiable instances much better than all other solvers (18 more than both the baseline Treengeling solvers), when using the pairwise ranking model.

5.2 Propagation-rate

We propose a new propagation rate-based splitting heuristic to improve the performance of divide-and-conquer parallel SAT solvers. We implemented our technique as part of the AMPHAROS solver [ALST16], and showed significant improvements vis-a-vis AMPHAROS on instances from the SAT 2016 Application benchmark. Our key hypothesis was that variables that are likely to maximize propagation are good candidates for splitting in the context of divide-and-conquer solvers because the resultant sub-problems are often simpler. An additional advantage of ranking splitting variables based on their *propensity to cause propagations* is that it can be cheaply computed using conflict-driven clause-learning (CDCL) solvers that are used as workers in most modern divide-and-conquer parallel SAT solvers.

In this section, we describe our propagation rate based splitting heuristic, starting with a brief description of AMPHAROS that we use as our base solver [ALST16]. We made our improvements in three steps: 1) We used Maplesat [LGPC16a] as the worker or backend solver. This change gave us a small improvement over the base AMPHAROS. 2) MapleAmpharos-PR: We used a propagation-rate based splitting heuristic on top of using Maplesat as a worker solver. 3) MapleAmpharos: We applied different restart policies at worker solvers of MapleAmpharos-PR.

5.2.1 The AMPHAROS Solver

AMPHAROS is a divide-and-conquer solver wherein each worker is a CDCL SAT solver. The input to each worker is the original formula together with assumptions corresponding to the path (from the root of the splitting tree to the leaf) assigned to the worker. The workers can switch from one leaf to another for the sake of load balancing and intensification/diversification. Each worker searches for a solution to the input formula until it reaches a predefined limit or upper bound on the number of conflicts. We call this the *conflict limit*. Once the conflict limit is reached, the worker declares that the cube[2] is hard and reports the "best variable" for splitting the formula to the master. A variable is deemed "best" by a worker if it has the highest VSIDS activity over all the variables when the conflict limit is reached. The Master then uses a load balancing policy to decide whether to split the problem into two by creating False and True branches over the reported variable.

5.2.2 Propagation Rate Splitting Heuristic

As mentioned earlier, the key innovation in MapleAmpharos is a propagation rate-based splitting heuristic. Picking variables to split on such that the resultant sub-problems are collectively easier to solve plays a crucial role in the performance of divide-and-conquer solvers. Picking the optimum sequence of splitting variables such that the overall running time is minimized is in general an intractable optimization problem.

For our splitting heuristic, we use a dynamic metric inspired by the measures that look-ahead solvers compute as part of their "look-ahead policy". In a look-ahead solver, candidate variables for splitting are assigned values (True and False) one at a time, and the formula is simplified against this assigned variable. A measure proportional to the number of simplified clauses in the resultant formula is used to rank all the candidate variables in decreasing order, and the highest ranked variable is used as a split. However, look-ahead heuristics are computationally expensive, especially when the number of variables is large. Propagation rate-based splitting on the other hand is very cheap to compute.

In our solver MapleAmpharos when a worker reaches its conflict limit, it picks the variable that has caused the highest number of propagations per decision (the propagation rate) and reports it back to the Master node. More precisely, whenever a variable v is branched on, we sum up the number of other variables propagated by that decision. The

[2] While the term cube refers to a conjunction of literals, we sometimes use this term to also refer to a sub-problem created by simplifying a formula with a cube.

propagation rate of a variable v is computed as the ratio of the total number of propagations caused whenever v is chosen as a decision variable divided by the total number of times v is branched on during the run of the solver. Variables that have never been branched on during the search get a value of zero as their propagation rate.

When a worker solver stops working on a sub-problem due to it reaching the conflict limit, or proving the cube to be UNSAT, it could move to work on a completely different sub-problem which has a different set of assumptions. Through this node switching, we do not reset the propagation rate counters.

The computational overhead of our propagation rate heuristic is minimal, since all the worker solvers do is maintain counters for the number of propagations caused by each decision variable during their runs. An important feature of our heuristic is that the number of propagation per decision variable is deeply influenced by the branching heuristic used by the worker solver. Also, the search path taken by the worker solver determines the number of propagations per variable. For example, a variable v when set to the value True might cause lots of propagation, and when set to the value False may cause none at all. Despite these peculiarities, our results show that the picking splitting variables based on the propagation rate-based heuristic is competitive for Application and cryptographic instances.

5.2.3 Worker Diversification

Inspired by the idea of using different heuristics in a competitive solver setting [HJS08], we experimented with the idea of using different restart policies in worker CDCL solvers. We configured one third of the workers to use Luby restarts [LSZ93], another third to use geometric restarts, and the last third to use MABR restarts. MABR is a Multi-Armed Bandit Restart policy [NLG+17], which adaptively switches between 4 different restart policies of linear, uniform, geometric and Luby. We note that while we get some benefit from worker diversification, the bulk of the improvement in the performance of MapleAmpharos over AMPHAROS and other solvers is due to the propagation rate splitting heuristic.

5.2.4 Experimental Results

In our experiments, we compared MapleAmpharos against 5 other top-performing parallel SAT solvers over the SAT 2016 Application benchmark and a set of cryptographic instances obtained from encoding of SHA-1 preimage attacks as Boolean formulas.

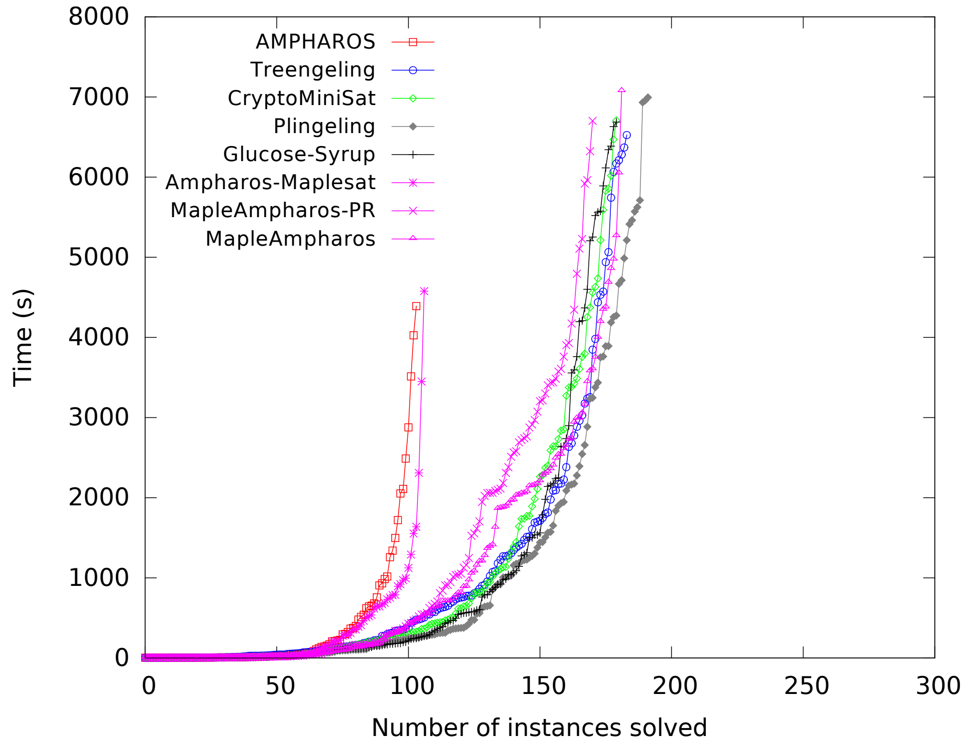


Figure 5.1: Performance of MapleAmpharos vs. competing parallel solvers over the SAT 2016 Application benchmark

Experimental Setup

We used the Application benchmark of the SAT competition 2016 which has 300 industrial instances obtained from a diverse set of applications. Timeout for each instance was set at 2 hours wall clock time. All jobs were run on 8 core Intel Xeon CPUs with 2.53 GHz and 8GB RAM. We compared our solver MapleAmpharos against the top parallel solvers from the SAT 2016 competition, namely, Treengeling and Plingeling [Bie16], CryptoMiniSat5 [Soo16], Glucose-Syrup [AS16] and also baseline version of AMPHAROS solver [ALST16]. Our parallel solver MapleAmpharos uses Maplesat [LGPC16a] as its worker CDCL solver.

Table 5.1: Solving time details of MapleAmpharos and competing parallel solvers on SAT 2016 Application benchmark

Solver	# of solved	SAT	UNSAT	Average Time (s)
MapleAmpharos	182	77	105	979.396
AMPHAROS	104	42	62	392.933
Ampharos-Maplesat	107	44	63	310.94
MapleAmpharos-PR	171	72	99	1035.73
CryptoMiniSat	180	72	108	942.894
Glucose-Syrup	180	74	106	898.767
Plingeling	192	76	116	965.167
Treengeling	184	77	107	969.467

Case Study 1: SAT 2016 Application Benchmark

Figure 5.1, shows the cactus plot comparing the performance of MapleAmpharos against the other top parallel SAT solvers we considered in our experiments. In this version of the MapleAmpharos solver we used the best version of worker diversification (which is a combination of Luby, Geometric and MAB-restart referred to section 5.2.3). As can be seen from the cactus plot in Figure 5.1 and the Table 5.1, MapleAmpharos outperforms the baseline AMPHAROS, and is competitive vis-a-vis Parallel CryptoMiniSat, Glucose-Syrup, Plingeling and Treengeling. However, MapleAmpharos performs the best compared to the other solvers when it comes to solving cryptographic instances.

Case Study 2: Cryptographic Hash Instances

We also evaluated the performance of our solver against these parallel SAT solvers on instances that encode preimage attacks on the SHA-1 cryptographic hash function. These instances are known to be hard for CDCL solvers. The best solvers to-date can only invert at most 23 rounds automatically (out of a maximum of 80 rounds in SHA-1) [NLG+17, Nos12]. Our benchmark consists of instances corresponding to a SHA-1 function reduced to 21, 22 and 23 rounds, and for each number of rounds, we generate 20 different random hash targets. The solution to these instances are preimages that when hashed using SHA-1, generate the same hash targets. The instances were generated using the tool used for generating these type of instances in SAT competition [Nos13]. The timeout for each

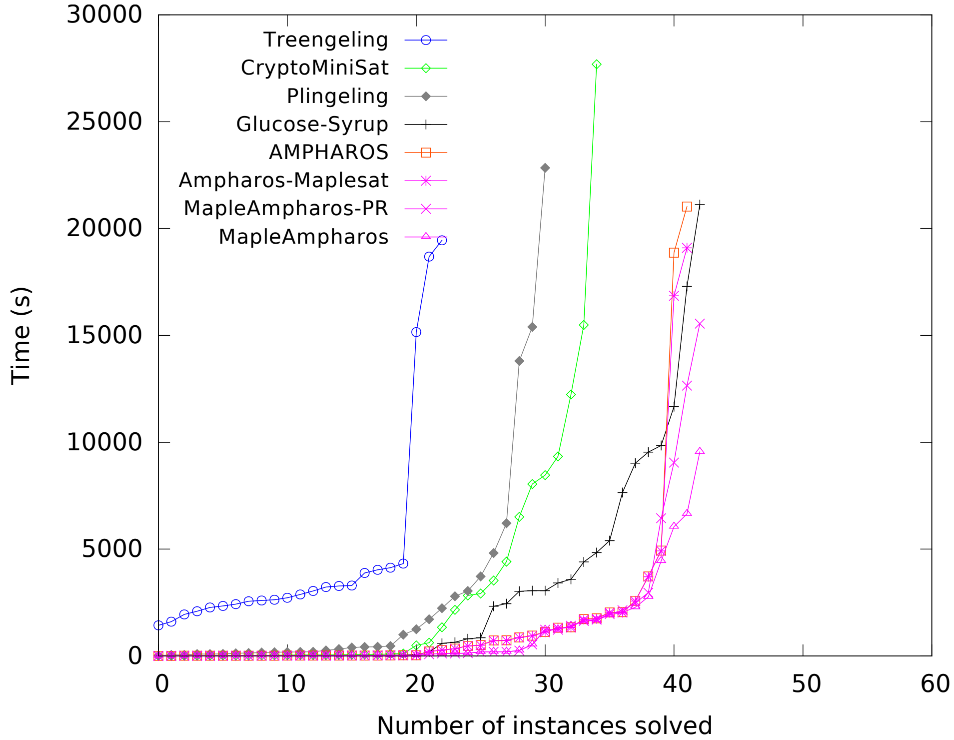


Figure 5.2: Performance of MapleAmpharos vs. competing parallel solvers on SHA-1 instances

instance was set to 8 hours. Figure 5.2 shows the performance comparison and Table 5.2 shows details of the average solving times on this benchmark. We compute the average for each solver only over the instances for which the resp. solvers finish. As can be seen from these results, MapleAmpharos performs the best compared to all of the other solvers. In particular, for the hardest instances in this benchmark (encoding of preimage attacks on 23 rounds of SHA-1), only Glucose-Syrup, AMPHAROS, and MapleAmpharos can invert some of the targets. Further, MapleAmpharos generally solves these SHA-1 instances much faster.

5.3 Machine Learning based Splitting Heuristics

In this work, we propose an ML-based method that takes as input features of a pair of variables of an input formula and statistics over solver state, and outputs ranking over

Table 5.2: Average solving time comparison on SHA-1 benchmark

Solver	# of solved	Average Time (s)
MapleAmpharos	43	1048.53
AMPHAROS	42	1619.1
Ampharos-Maplesat	42	1518.76
MapleAmpharos-PR	43	1457.14
CryptoMiniSat	35	3056.31
Glucose-Syrup	43	2912.84
Plingeling	31	2668.48
Treengeling	23	4783.35

this pair of variables. This ML-based "comparator" is in turn used by our DC solver to rank variables for splitting at regular intervals during its run. We implemented our heuristics in the Painless parallel solver framework [LFBSK17] (we refer to our solver as MaplePainless), and compared it with top parallel SAT solvers from the recent SAT competitions. We find that our ML-based method out-performs the best DC solvers on both SAT 2018/2019 competition as well as cryptographic instances. We only compare our MaplePainless solver against the state-of-the-art DC solvers, because DC solvers are known to perform poorly relative to portfolio solvers over application instances.

In this section we discuss a formulation of the splitting problem, define a quality measure for splitting, and study how we can train ML models that approximate the best splitting variable.

5.3.1 The Splitting Problem

Given a Boolean formula ϕ, a sequential solver S, and performance metric pm, the splitting problem is to determine a variable v in ϕ such that the time required to solve each of $\phi[v = T]$ and $\phi[v = F]$ by S is minimal over all variables in ϕ with respect to the given performance metric pm, i.e. to find $\text{argmin}_{v \in vars(\phi)}\{pm(\phi, v)\}$.

Modeling the exact behavior of a DC solver as it solves the sub-formulas in parallel and splits them on demand, is a challenging task. Below we define a metric that we believe is a more accurate measure of the optimal choice of a splitting variable, compared to the heuristic metrics mentioned in Section 2.4.

Let $\phi_1 = \phi[v = F]$ and $\phi_2 = \phi[v = T]$ be sub-formulas of splitting ϕ over v, and let $t_1 = t_S(\phi_1)$ and $t_2 = t_S(\phi_2)$ be runtimes of solving them by sequential solver S. The total time taken to solve the formula ϕ in this setting depends on the status and runtimes of the sub-formulas. If ϕ is UNSAT, the solver needs to prove both of the sub-formulas UNSAT. Hence the total time to solve such an instance is the maximum of the solver runtimes over the two sub-formulas. If on the other hand the formula ϕ is SAT, at least one of the sub-formulas must be SAT. If both sub-formulas are SAT, the total time is the minimum of the two, otherwise, only the SAT sub-formula matters. The total time of solving ϕ after splitting over variable v can be represented as follows:

$$T_{total}(\phi, v) = \begin{cases} \max(t_1, t_2), & \phi_1 : UNSAT, \phi_2 : UNSAT \\ t_2, & \phi_1 : UNSAT, \phi_2 : SAT \\ t_1, & \phi_1 : SAT, \quad \phi_2 : UNSAT \\ \min(t_1, t_2), & \phi_1 : SAT, \quad \phi_2 : SAT \end{cases}$$

We use this total runtime as our performance metric: $pm(\phi, v) = T_{total}(\phi, v)$. In other words the target of our splitting heuristic is: given formula ϕ, find a variable $v = \text{argmin}_{v \in vars(\phi)}\{T_{total}(\phi, v)\}$.

5.3.2 Handling Timeouts

In practice, sub-formulas obtained after splitting on a variable can be hard for SAT solvers and thus they may timeout for those cases. Let the status of a timed out (sub-)formula be labeled as "UNKNOWN". For a pair of variables u and v in formula ϕ, we collect the runtime and status of solving sub-formulas $u_1 = \phi[u = F], u_2 = \phi[u = T], v_1 = \phi[v = F]$ and $v_2 = \phi[v = T]$. If the status of all four of these sub-formulas are UNKNOWN, we cannot derive the truth label (we do not know which of these two variables is better for splitting). In all other cases (mix of having SAT/UNSAT and UNKNOWN), we have enough information to be able to compare u and v.

5.3.3 Learning to Rank

Generally, performance metrics can be used to generate a total order over the splitting variables (the higher ranked variables have a higher performance metric). Thus we can see the splitting problem as picking the minimum element from a ranked list. A common way in implementing splitting heuristics is to rank the variables by directly deriving the

performance metric of each variable and selecting the minimum element. However, this is not the only way one can rank the elements. There are three main approaches in the ML literature for learning a model to rank a list of elements [L+09]:

- **Pointwise**: Learning a numerical or ordinal score for each data point, which are in turn sorted according to their ordinal score. The problem here translates to training a regression model.
- **Pairwise**: In this approach, ranking is done via learning a model that acts as a comparator, which takes as input two data points and output a total order over them.
- **Listwise**: These algorithms try to directly minimize a ranking evaluation metric (e.g. τ-score or Mean Average-Precision) that compares a predicted ranking against a true ranking.

Almost all previous branching and splitting heuristics use pointwise ranking. For example, VSIDS branching heuristics [MMZ+01] maintains a score for each variable, which represents how much that variable participated in clause learning recently. Then the variable with the highest activity is picked. Ultimately, the goal is to minimize the runtime and one might learn a function that directly approximates the desired runtime based ranking. However, approximating the runtime distribution of the CDCL SAT solver is very hard in general, as the interplay of the many heuristics in CDCL solvers makes it hard to predict how the search progresses. The hope of heuristic designers is that their variable ranking strongly correlates with a ranking where high ranking variables generate easier sub-formulas. In other words, their variable ranking using the proxy metric strongly correlates with runtime-based ranking. In the case of splitting or branching heuristics, we do not care about the actual runtime of sub-formulas and only want to know which variable corresponds to the lowest runtime. In other words, we want a way of comparing runtimes and not exactly deriving the runtime values. As mentioned above, we are looking for a minimum element in an array, sorted based on a metric. We approach this task of finding the minimum using two different methods. First, we build a *pairwise ranking* model that learns to compare two elements (two variables in our case), and second, we use a modified version of ordinal ranking, that we call *min-rank*, where we build a classifier that determines whether a given variable sits at the rank 1. We use+d binary classification for building both of these models. In the pairwise ranking, we use the model as a less-than operator and find the minimum in a linear scan. In min-rank, we check all of the variables against the model and pick the variable that the model declares as the minimum.

The first model is represented by a binary classifier PW (PairWise) that takes as input features of a formula ϕ and features of two variables v_i and v_j within ϕ, and answers the question of "is v_i better than v_j for splitting ϕ?" (according to our splitting performance metric described in Section 5.3.1).

$$PW(\phi, v_i, v_j) = \begin{cases} 1, & pm(\phi, v_i) < pm(\phi, v_j) \\ 0, & otherwise \end{cases} \tag{5.1}$$

This type of predicate learning was also used in one of the SATZilla versions [XHHLB12] (known as pairwise voting), to rank a list of algorithms on a given instance.

For the second model, we used the idea of reduction by Lin et al. [LL12] for implementing ordinal ranking using binary classification. In their work, the role of a binary classifier given an element and an integer rank k is to determine whether the element is within the top k elements or not. Splitting heuristics look for the top variable in a ranked list, thus we are only interested in the $k = 1$ case. We define a binary classifier MR (Min-Rank) that takes as input a variable v, and answers the question "is v the best variable for splitting ϕ?".

$$MR(\phi, v_i) = \begin{cases} 1, & \forall j \neq i : pm(\phi, v_i) < pm(\phi, v_j) \\ 0, & otherwise \end{cases} \tag{5.2}$$

5.3.4 Features for Training the Models

The data points that we used to train the model have the following format:

$$\begin{array}{ll} PW: & (\langle formula_{features}(\phi), var_{features}(v_i), var_{features}(v_j)\rangle, \{0,1\}) \\ MR: & (\langle formula_{features}(\phi), var_{features}(v)\rangle, \{0,1\}) \end{array} \tag{5.3}$$

where the last element corresponds to the appropriate classifier ($PW(\phi, v_i, v_j)$ or $MR(\phi, v)$). For the formula features, we started from the features proposed by SATZilla in SAT competition 2012 [XHHLB12]. Compared to the model that has been used in SATZilla, we will query our model at each splitting point. The feature computation time can quickly become a big part of the total runtime, and dominate the gain from picking a better splitting variable. On the other hand, each of the features could have an important role in making the model representative of the target distribution. To address this problem we performed a feature selection on our initial set of features (both formula and variable features). We first

Table 5.3: Variable ($var_{features}(v)$) and Formula features ($formula_{features}(\phi)$).

Feature name	Description
activity	VSIDS activity [MMZ+01]
LRBProduct	product of LRB [LGPC16a] activities of v and $\neg v$ literals
numFlip	#times the implied value of v is different than its cached value [AHJP14]
propRate	average #propagation over #decision [NNS+17]
numDecided	#times v has been picked in branching
numAssigned	#times v got a value through branching/propagation
numLearnt	#times v appeared in a conflict clause
decisionLevel	average of decision levels of v at the end of the limited search
numInBinary	#times v appears in a clause of size 2
numInTernary	#times v appears in a clause of size 3
numDecisions	number of decisions made in the limited search
numPropagations	number of unit propagations in the limited search
conflictRate	ratio of #conflict clauses over #decisions
totalReward	sum of LRB reward of all of the variables
numBinary	number of clauses of size 2 in ϕ
numTernary	number of clauses of size 3 in ϕ
avgVarDegree	average variable node degree in the Variable-Clause graph
avgClauseDegree	average clause node degree in the Variable-Clause graph

removed the very heavy features like LP-based (linear programming) features. We used random forest for training our models. We then extracted the relative importance of each feature after training, which corresponds to the frequency of appearance of those features in the ensemble of decision trees. We created a sorted list of features based on their relative importance (f), and performed a forward feature selection [GE03]. More specifically, starting with an empty list F, we passed through f and added the features to F, if they reduced the cross-validation error when training on F. We then performed a pass on F, to remove heavy-to-compute features that do not contribute much to the accuracy of the model. We also took into account the product features (features from the multiplication of pairs of other features) to add non-linearity to the model. The final variable and formula features are listed in Table 5.3, consisting of structural metrics and metrics from a limited search.

5.3.5 Training Data

We used the MapleCOMSPS solver [LOG+17] for solver runtime and formula/variable feature collection. For generating our training data set, we picked 210 instances from the collection of application/crafted benchmarks of SAT competition 2016 and 2017 [HJT16, HJT17]. To be more precise, 87 instances from the application benchmark of 2016, 21 instances from the crafted benchmark of 2016, and 102 instances from the main benchmark of 2017. The selection criteria were based on having instances from different types of problems (not problems of the same kind with different sizes) and having a wide range of hardness to make a representative training set. We did not use any instance that was deemed too hard (timed out) or too easy (was solved under 5 seconds) by our sequential solver. To match the test environment, we first ran the pre-processing stage of MapleCOMSPS and simplified the formulas. Then we computed all of the structural formula features offline and for the search probing features we ran MapleCOMSPS up to 10,000 conflicts and collected the necessary statistics from the solver. For computing the true labels, we randomly selected 50 variables in each instance and split the formula on each of them and solved the sub-formulas with MapleCOMSPS up to a 5000 seconds timeout, recording the runtime and status (SAT, UNSAT, UNKNOWN).

5.3.6 Analysis of the Learned Models

For training the model, we used *random forest classifier*. We can achieve an average precision of 83% and an average recall of 83%, and accuracy of 80.7%. The candidate variable list can be ordered using the learned predicate. For finding the best variable, we only need to find the "min" of the list, which can be done in linear time. Although, when using a noisy comparator, the error caused by the inaccurate comparison, might accumulate over multiple comparisons. There are more robust sorting algorithms in the presence of noisy comparators (e.g. counting method [SW17]), but the running time complexity is quadratic in the number of elements, which is not feasible for large formulas. To check how well our predicate is performing, we ranked the variables in the instances in our training set, where we have the true labels.

When sorting the variables using the pairwise ranking, out of 210 instances, in 120 instances the best variable in the predicted ranking matched the actual best variable (57.1% of the time). In 18 cases the best actual variable was the second best predicted variable. The worst prediction happened in an instance with 2200 variables, where the best actual variable appeared in 30th position in the predicted list. The total error (e.g. τ score) of comparing the predicted ranking and the best actual ranking could be poor, however, we

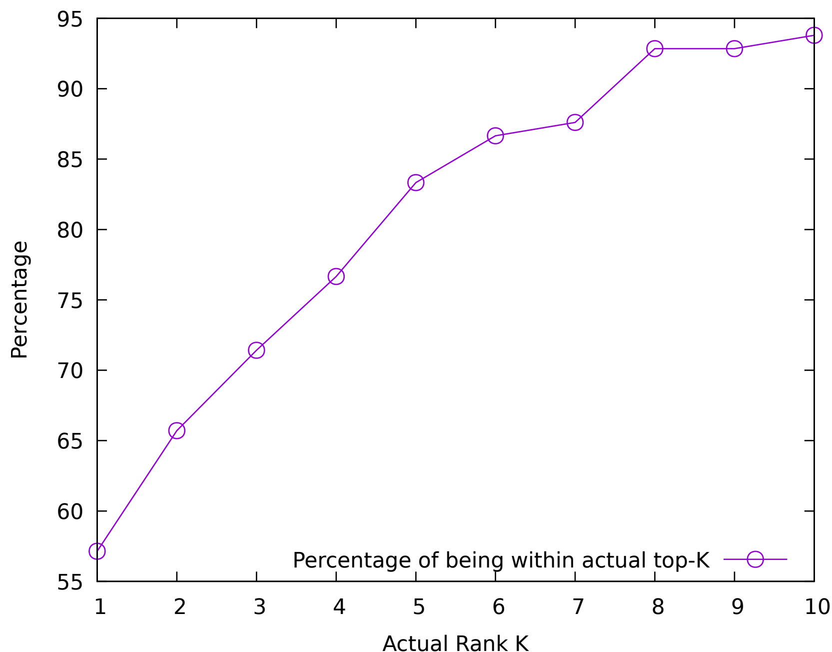


Figure 5.3: Percentage of instances where the predicted best variable is within the actual top-k variables for k between 1 and 10.

can see some general ordering over the variables (variables that are much better choices appear closer to the front of the list). Figure 5.3 shows the percentage of instances (out of 210), where the predicted best variable (the output of the model for splitting), is within the actual top-k variables. We observed that the best predicted variable is one of the actual top 10 variables in 197 out of 210 instances (93.8%). This shows that top variables in our predicted ranking have a considerable overlap with the top variables in the actual ranking, although not appearing in the same exact order.

Both of the pairwise and min-rank models do a linear scan of the variables to find the best splitting variable. The worst case time complexity of them is $O(T_C \cdot n)$, where n is the number of variables and T_C is the time complexity of querying each of the classifiers. The best case time complexity of pairwise ranking is $\Omega(T_C \cdot n)$ as well, because the best variable candidate must be compared with all others, however, the best case for min-rank is to hit the best variable in the first step, which gives $\Omega(T_C)$ complexity.

5.4 Implementation

Our implementation of MaplePainless is built using the Painless solver framework [LFBSK17]. Painless is a state-of-the-art framework that allows developers to implement many different kinds of parallel SAT solvers for many-core environments. The main configurable components of Painless are: parallel strategies such as DC or portfolio, clause sharing and management policies, and diverse sequential engines. The implementation of our machine learning based splitting heuristic relies on the use of the DC strategy in Painless [LFBSK19]. We use an instrumented version of the MapleCOMSPS [LOG+17] solver as workers in MaplePainless. The instrumentation collects formula/variable statistics and chooses splitting variables.

5.4.1 Implementation of Splitting in Painless-DC

As discussed earlier, Painless-DC splits a formula at regular intervals throughout its run. At a high-level, the master node maintains a queue of idle cores to assign jobs to. Initially, the master node chooses a variable to split and assigns the resultant sub-formulas to two cores. If the queue of idle cores is non-empty, the master node chooses a sub-formula from one of the busy cores and splits it into two sub-formulas, one of which is assigned to the busy core and the other to one of the idle ones. This process is repeated until the queue of idle cores is empty. If during the solver's run a core becomes idle and is added to the idle queue (e.g., if it has established UNSAT for its input sub-formula), the above-mentioned process is invoked until the idle queue becomes empty again. This form of load-balancing ensures that worker nodes are not allowed to idle for too long.

5.4.2 Feature Computation in MapleCOMSPS for Machine Learning

When it is time to split a formula, Painless' master node asks the sequential worker solver whose sub-formula is being split for variables to split on. The worker solver computes formula and variable features (e.g. number of times a variable is assigned, either decided or propagated) on the sub-formula to be split. Majority of the variable features are dynamic and their counters are updated whenever there is a related action performed during the search, thus their complexity is amortized over the run of the solver. The description of the variable features is listed in Table 5.3.

For the machine learning models PW and MR, we use the random forest classifier, because it gave us better accuracy in the training phase. We used scikit-learn python package [PVG+11] for training the model and used scikit-learn-porter [Mor] to generate a C code out of the trained model. The trained weights are embedded in the exported C code. We later call this code from MapleCOMSPS for performing predictions. Both the pairwise classifier PW and the min-rank classifier MR are iteratively called to identify minimum ranked variable over all the variables of the input formula. Note that the ML models are noisy, i.e., their ranking is not likely to correspond exactly to the true ranking according to our performance metric. However, the accuracy of the ML models is high enough to be acceptable in our setting.

5.5 Experimental Results

Here we present experimental results comparing our ML-based heuristics in MaplePainless (described in Section 5.3) against the baseline Painless and state-of-the-art `Treengeling` solvers.

5.5.1 Evaluation over SAT 2018 and 2019 Competition Instances

Experimental Setup.

For evaluation, we used the main track benchmark of the SAT competition 2018 [HJS18a] and SAT race 2019 [HJS16], which in total have 800 instances, consisting of industrial instances coming from a diverse set of applications and crafted instances encoding combinatorial problems. Timeout for each instance was set at 5000 seconds wallclock time (the same as in SAT competitions). All jobs were run on Intel Xeon CPUs at 3GHz and 64GB of RAM.

Solvers Description.

We compared our solver against the top divide-and-conquer parallel solvers, Treengeling [Bie17] version bcj and Painless-DC [LFBSK19] with its best performing setting (node switch strategy: *clone*, clause sharing: all-to-all, and splitting heuristic: *flip*), which we will refer to as `Treengeling` and `Painless-Flip`, respectively. We refer to our implementations using the PW classifier for pairwise ranking as `MaplePainless-Pairwise` and the one

Table 5.4: Performance comparison of our solvers vs state-of-the-art divide-and-conquer parallel SAT solvers. Number of solved instances in each benchmark is out of 400, and for Total row, it is out of 800. SAT column shows the number of satisfiable instances solved (resp. UNSAT). The bold entries, show the best result on benchmark in each column.

Benchmark	Solver	#Solved	SAT	UNSAT	Avg. Runtime	PAR-2
SAT 2018	Treengeling	245	143	102	827.097	486.844
	Painless-Flip	244	144	100	**384.888**	459.42
	MaplePainless-MinRank	**255**	**150**	**105**	507.399	**438.719**
	MaplePainless-Pairwise	251	149	102	392.878	441.281
SAT 2019	Treengeling	**259**	150	**109**	617.665	436.104
	Painless-Flip	234	149	85	500.185	493.623
	MaplePainless-MinRank	246	151	95	**466.044**	459.624
	MaplePainless-Pairwise	254	**162**	92	485.218	**439.790**
Total	Treengeling	504	293	**211**	719.472	922.948
	Painless-Flip	478	293	185	441.330	953.043
	MaplePainless-MinRank	501	301	200	487.092	898.343
	MaplePainless-Pairwise	**505**	**311**	194	**439.322**	**881.071**

with MR classifier for binary classification of minimum rank as MaplePainless-MinRank. Our parallel solver uses MapleCOMSPS [LOG+17] as the backend sequential solver. We changed MapleCOMSPS to always use LRB as branching heuristics. Each solver was assigned 8 cores.

Results

To perform an apple-to-apple comparison and measure the effectiveness of our splitting heuristics, we reused all of the configurations and components of Painless-Flip and only replaced the splitting heuristics, which was straightforward, thanks to the modular design of Painless. Table 5.4 lists the number of solved instances, average runtime among solved instances and the PAR-2 metric. In the SAT competition, PAR-2 is measured in seconds, but for better readability, we report it in hours. As the table shows, both machine learning based heuristics, MaplePainless-Pairwise and MaplePainless-MinRank, improve significantly upon the baseline in both SAT 2018 and SAT 2019 benchmarks. They also solve more instances than Treengeling in SAT 2018 benchmark. Although solving fewer instances in SAT 2019. Additionally, MaplePainless-Pairwise solves more instances than Treengeling and has the lowest PAR-2 score among all in the combined benchmarks of 2018 and 2019. Figures 5.4 and 5.5 show the cactus plots of the proposed, baseline and state-of-the-art solvers over the SAT 2018 and SAT 2019 benchmarks. Treengeling per-

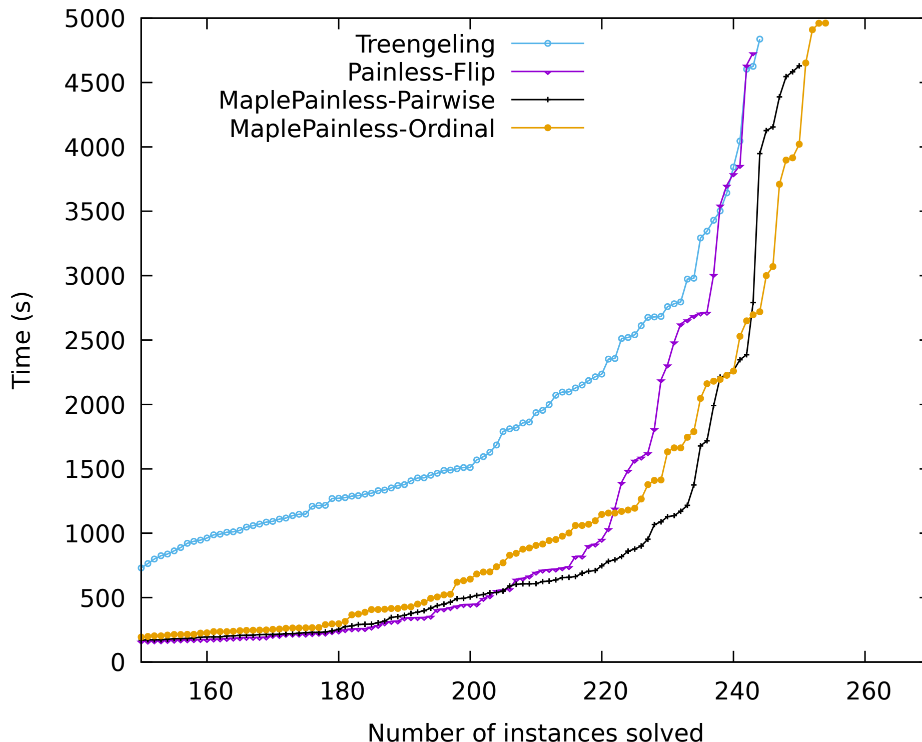


Figure 5.4: Cactus plot for performance comparison of parallel SAT solvers on filtered main track benchmark of SAT 2018.

forms a variety of (in-)processing before the search and after each splitting, which is a powerful component of this solver. To evaluate the contribution of search space splitting to the number of solved instances, we ran `Treengeling` and MaplePainless-`Pairwise` with the simplification routines turned off. `Treengeling` solves 493 instances (out of 800) and MaplePainless-`Pairwise` solves 495 instances on the same benchmark.

SAT vs. UNSAT

From Table 5.4 we can observe that both of the ML-based solvers are better at solving satisfiable instances rather than unsatisfiable instances. For example, MaplePainless-`Pairwise` solves 18 more satisfiable instances compared to `Treengeling`, while `Treengeling` solves 17 more unsatisfiable instances, although with higher average runtime and PAR-2.

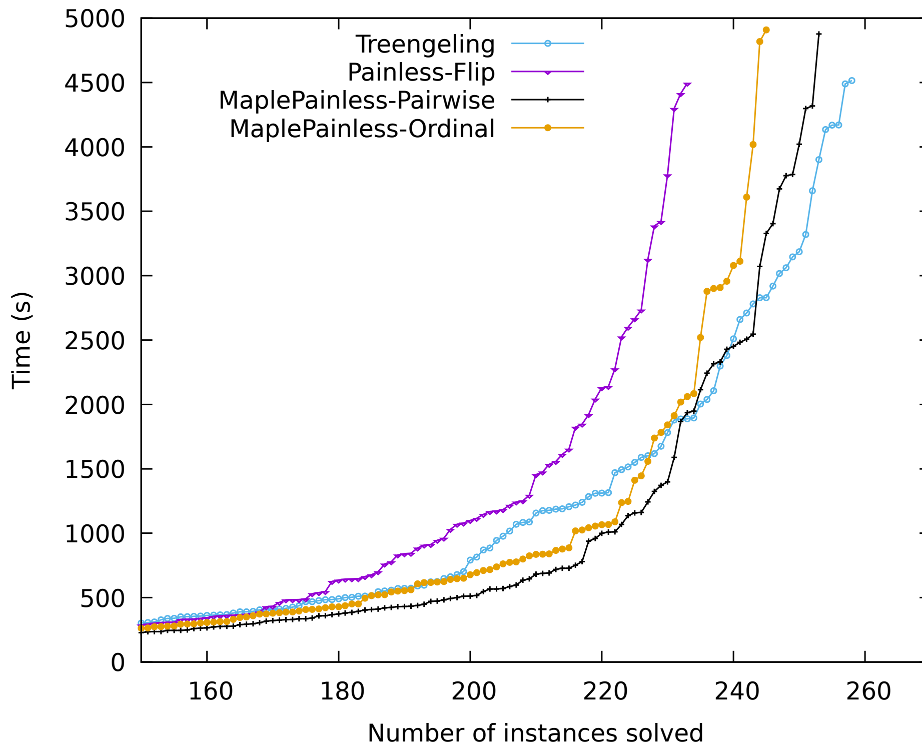


Figure 5.5: Cactus plot for performance comparison of parallel SAT solvers on filtered benchmark of SAT race 2019.

5.5.2 Evaluation over Cryptographic Instances

Experimental Setup

We used a set of hard cryptographic instances encoding preimage attack on round-reduced SHA-1 hash function. More precisely, the instances encode inversion of 21, 22 and 23 rounds SHA-1, with 20 random targets for each rounds version [NHG+17]. All jobs were run on Intel Xeon CPUs at 3GHz and 64GB of RAM with 12 hours wallclock timeout.

Solvers Description

We compared our MaplePainless-Pairwise solver against the baseline (Painless-Flip) and Treengeling. All solvers were run with 8 cores. For the backend solvers in this experiment, we used Glucose [AS18] and MapleSAT [LGPC16a] (4 of each). Glucose

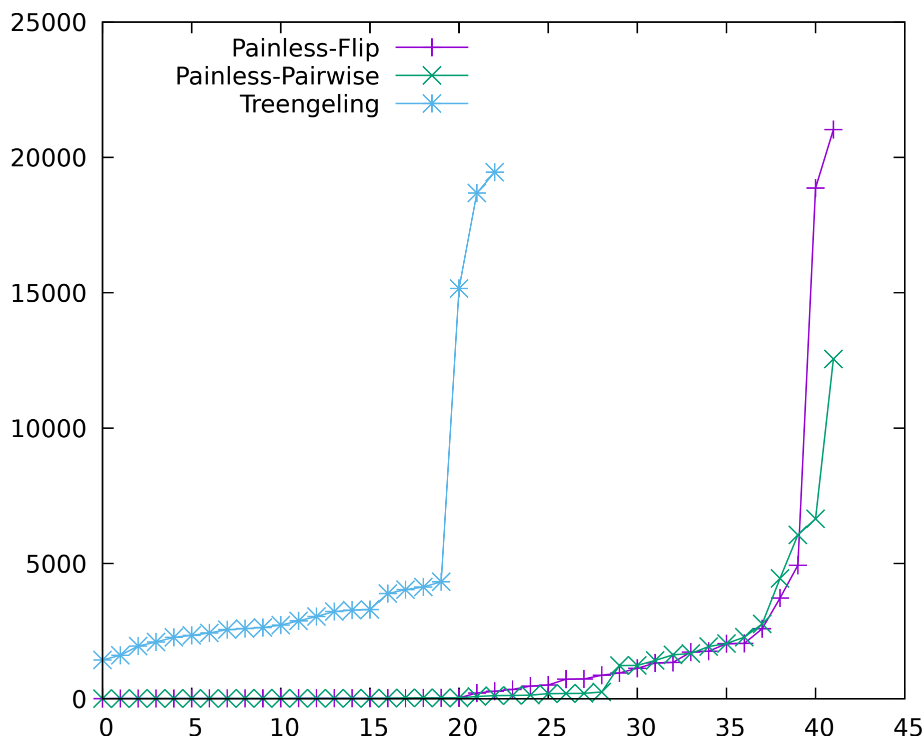


Figure 5.6: Performance of MaplePainless-Pairwise against baseline Painless and Treengeling on cryptographic instances.

solvers used the Glucose's default restart policy. MapleSAT solvers were set to use the MABR restart policy [NLG+17]. To have an apple-to-apple comparison with baseline, we used the same backend solver configuration for baseline and our solvers.

Results

Figure 5.6 shows the performance of the considered DC solvers on our hard cryptographic benchmark. Instances with 21 rounds are easy for all solvers. 22 rounds instances are much harder than 21 rounds instances and as can be seen, Treengeling solves very few of these instances. Although both MaplePainless-Pairwise and Painless-Flip solve all of these instances. The hardness ramps up very quickly at 23 rounds instances, where Treengeling does not solve any of the instances and Painless-Flip solves 2 of them. MaplePainless-Pairwise solves 3 instances in this subset of instances, and with 30%

lower runtime.

5.6 Related Work

Cube-and-conquer [HKB18] solvers (such as Treengeling [Bie17]) use a look-ahead procedure to determine the best splitting variable. Look-ahead techniques check for each variable, what will happen if that variable is picked for splitting and measure the difference between original formula and generated sub-formulas. This measure will be used as a score for each variable to give a ranking of the candidates. Checking all of the variables can be very expensive in large formulas, thus the solver might choose to run this procedure only on a subset of variables. The subset is chosen according to a pre-selection heuristics. Also use of clever data structures like tree look-ahead can speed up the process significantly.

In contrast to look-ahead techniques, some solvers look back at previous search and formula statistics to identify the best candidate at the current splitting point. Ampharos [ALST16] picks the variable with the highest VSIDS activity and MapleAmpharos [NNS+17] uses propagation-rate (average propagation over decision). In [AHJP14], the number of times a variable's saved phase is flipped through propagation is used as a measure of predicting how much that variable could affect other variables when set in both splitting branches. This has been shown to be effective in divide-and-conquer settings [LFBSK19]. We can categorize our work as a look-back heuristic as all of the features are extracted from the previous limited runs.

Machine learning has been used to rank and pick the best variable in sequential SAT solvers. Liang et al. use a reinforcement learning formulation to find the most rewarding variable according to the learning-rate metric [LGPC16a]. In another work, they train a logistic regression model that ranks variables based on the probability of causing a conflict in the next step [LVP+17]. These models similar to the majority of variable order heuristics follow a pointwise ranking method (*i.e.,* learning a score for each variable and picking the variable with highest score). However, we are employing a pairwise ranking method.

The pairwise ranking has been used in other constraint programming contexts as well. Xu et al. used pairwise voting in the context of algorithm selection, to rank SAT solvers based on their performance on a single formula [XHHLB12]. Khalil et al. formulated branching in Mixed Integer Programming and used SVM^{rank} to optimize for number of ranking inversions [KLBS+16].

5.7 Chapter Summary

We presented a propagation rate-based and a machine learning based splitting heuristic for divide-and-conquer solvers. Our propagation-rate heuristic, implemented on top of AMPHAROS, outperform all other considered divide-and-conquer and portfolio solvers on cryptographic instances, and is competitive with top solvers on application instances. Most of the branching/splitting heuristics in sequential and parallel SAT solvers rank the variables based on assigning scores to each variable (pointwise ranking). We took a step back and looked at different approaches to the ranking problem itself and studied pairwise ranking and ordinal ranking. In both cases, we trained binary classifiers that act as predicates. In the pairwise ranking, the predicate is a less-than operator that can sort the variable list according to their quality, and in the ordinal ranking, the predicate can mark variables that their rank is higher than a threshold, and thus be used to find top k variables. Our approximated ranking either with pairwise or ordinal ranking can identify the actual top 10 variables (according to our model) with a high probability, although not in the same exact order. One of the challenges in this work was the computation of features extracted from the formula and variables. We performed a feature selection process in a way that reduces the feature computation time without sacrificing the model's precision. We evaluated our implementation in Painless framework against top divide-and-conquer parallel SAT solvers. We performed an apple-to-apple comparison with `Painless-Flip`, by only replacing the splitting heuristic and keeping the rest of the modules and configurations the same. We were able to improve upon `Painless-Flip` by solving more than 10 additional instances in each of the main track benchmarks of SAT competition 2018 and SAT race 2019. The `MaplePainless-Pairwise` solver can solve 6 more instances compared to `Treengeling` on SAT 2018 benchmark, however, `Treengeling` solves 5 more instances than `MaplePainless-Pairwise` on SAT 2019 benchmark. Furthermore, we are solving significantly more instances than `Treengeling` on cryptographic benchmark and faster than the baseline.

Chapter 6

Conclusion

In this thesis, we have studied SAT-based cryptanalysis of standard cryptographic hash functions in SHA-1 and SHA-2 family. Our contributions were in lines of extending/enhancing reasoning components, search heuristics, and pre-processing, in the context of a divide-and-conquer parallel SAT solvers with CDCL backend solvers. In this chapter, we highlight our main results and the impact of each contribution and point out possible future directions for this work.

6.1 Overview of Results

Briefly, the overview of our results in this dissertation are:

- Chapter 3: We developed a SAT+Crypto system which we called CDCL(CRYPTO), that extends the propagation (implication) and conflict analysis (learning) components of a CDCL SAT solver using a programmatic SAT architecture. These programmatic extensions can implement any cryptanalytic or problem constraints that are too heavy to be encoded into CNF and are better to be queried lazily. We implemented a programmatic version of MapleSAT and used it to improve algebraic fault attacks on hardware implementations of SHA-1 and SHA-256, recovering the secret message bits with fewer required faults compared to the state-of-the-art algebraic and differential fault attacks on these functions. Furthermore, we used this system to improve the state-of-the-art SAT-based collision finding of SHA-256, by one round.

- Chapter 4: We used Bayesian moment matching formulation of the Boolean SAT problem and turned it into a pre-processing technique with local updates to arrive at a promising starting point for polarity and activity initial values. We also incorporated a corrective feedback loop to update the initial point using the clause learning routine of a CDCL SAT solver. We improved runtime of SHA-1 preimage attack instances by 4x on average, and further demonstrated the effectiveness of this method on application instances compared to other initialization methods in a set of leading CDCL solvers in their respective benchmarks.

- Chapter 5: We formulated splitting heuristic of divide-and-conquer solvers as a ranking problem. We presented an ad-hoc metric (propagation-rate) for pointwise ranking. We then proposed a runtime-based metric for the quality of splitting variables and presented two machine learning models for pairwise and ordinal ranking that maps static and search features of a formula and variable within that formula to a ranking of those variables. These models were trained offline with the target of giving the variables with the higher splitting quality a higher rank. We improved the baseline parallel solvers (AMPHAROS and PAINLESS) in terms of runtime on cryptographic instances and solved instances over application instances. Also beating the state-of-the-art divide-and-conquer solver from the recent SAT competitions.

6.2 Impact and Takeaways

The main takeaway of our CDCL(CRYPTO) work is that the black-box use of SAT solvers in cryptanalysis (only focusing on better CNF encodings), has an implicit cap on the benefits that we could harness from the search capabilities of SAT solvers. The two main lines of specializing a solver to a class of problems are tailoring heuristics and extending reasoning components. While both are crucial in getting speedups in orders of magnitude, we believe that the fundamental capabilities of extended reasoning are far more important. Furthermore, the CDCL(CRYPTO) framework allows for implementing any cryptanalysis technique (e.g. differential or linear cryptanalysis). Therefore we position our work as the potential next step toward better cryptanalysis tools that are flexible and at the same time harness the search power of SAT solvers.

The second takeaway of our work is within our parallel SAT solver, where unlike usual heuristic designs that try to approximate the value of a variable individually, we stepped back and looked at the ranking problem itself. This allowed us to approach the problem differently and seek other ways of tackling the problem, which led us to employ existing

ideas in the literature for ranking a list of candidates. Our approach tries to capture a wide variety of applications and is not limited to cryptographic instances.

Regarding the Bayesian moment matching based heuristics, we took a fresh look at the initialization problem in the SAT solvers that have been less explored compared to the other components of the CDCL SAT solvers. We showed that for satisfiable instances this initialization method can give a very promising starting point.

6.3 Limitations

In this section, we discuss some of the known limitations of our work.

6.3.1 Programmatic Extension of Reasoning Components

The programmatic callback mechanism can be seen as an abstraction-refinement process, where the parts of input problem that are costly to encode in CNF (e.g. encoding of the higher level logic properties), are abstracted from the CNF formula and are added back in forms of programmatic reason clauses and programmatic conflict clauses. For the types of problems that the solver needs to refine back most of the abstracted away formula, the programmatic approach might not outperform encoding all of the implications in CNF from the start (e.g. parity function).

6.3.2 Machine Learning based Splitting Heuristic

The main limitation of this work is on modeling the divide-and-conquer splitting tree. Indeed, capturing the exact behavior of a divide-and-conquer solver, building and navigating a splitting tree, is a very challenging task. Therefore we modeled a very simplified version of these solvers, where the formula is split once and the two sub-formulas are solved in parallel independently of each other. There are two challenges with this model. The first challenge is that in practice all of the worker solvers share information with each other, and the shared information (learned conflict clauses) affects workers' runtime. The second challenge is that in practice the splitting process will be performed several times and that is how the splitting tree is created. However, to address the second challenge, we can see the process as split once and solve the two sub-formulas using two parallel solvers instead of sequential solvers. With the assumption that those parallel SAT solvers are at least

as good as the sequential ones, we can see that the second challenge imposes an inherent bound on getting to the optimal splitting (i.e. it will not necessarily find the optimal variable).

We performed supervised learning, and the learning process was done offline on a collection of existing instances. Although the accuracy of our models in our limited setting is relatively good, they are geared more toward solving instances of similar structure and might not be suitable for instances from different classes. Approaches like LRB [LGPC16a] are adaptive and move with the changing state of the solver. However, the splitting problem needs a more global view rather than focusing on local sub-spaces (as in CDCL), therefore an adaptive approach would have limited learning abilities in this setting.

6.3.3 Heuristic Initialization using Bayesian Moment Matching

The main limitation of analyzing the input SAT formula using Bayesian moment matching for the heuristic initialization is that the BMM model assumes that the problem is satisfiable. The posterior distribution might not be very fruitful for unsatisfiable instances. Although our main focus in this dissertation is on the class of cryptographic instances that are all satisfiable, having an initialization that works equally well on unsatisfiable instances would be very valuable.

Another limitation of the BMM approach is the problem of scaling to very large instances. Formulas with a large number of clauses, need more epochs (number of passes over the set of clauses) to converge to a promising starting point. However, at the same time, each epoch takes more processing time, simply because there are more clauses to process. In a setting with a fixed CPU time budget for pre-processing, BMM might not be able to converge to a promising starting point for instances with tens of millions of clauses. Note that, this was not a problem in our cryptographic instances, as they are relatively small compared to industrial instances, and we afforded to perform in the order of 100 epochs (compared to 10 epochs on industrial instances).

6.4 Future Work

Here, we outline the following questions for future work:

- *Can we use the pairwise and ordinal ranking (min-rank) in branching heuristics for sequential solvers as well?*

Conceptually the branching heuristics also rank the variables and pick the best variable according to some metric. Therefore the pairwise or min-rank type ranking can be adapted to branching heuristics as well. The main challenge would be the overhead of feature computation and querying the ranking models, as branching heuristics routines are invoked much more than splitting heuristics routines during the run of a CDCL SAT solver.

- *Can* CDCL(Crypto) *be applied on other cryptographic primitives?*
  The design of CDCL(Crypto) is fairly general and the programmatic callbacks can be used for encoding different cryptanalytic techniques. For instance, ARX based block ciphers can be a good target as they have similar building blocks to SHA cryptographic hash functions.

- *What are other potential use cases of BMM?*
  Because BMM is essentially trying to maximize the number of satisfied clauses, it can be used in MAX-SAT settings as well, and maybe in conjunction with a CDCL solver for Partial-MAX-SAT formulas.

References

[ADN+10] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: On critical paths and clock faults. In *International Conference on Smart Card Research and Advanced Applications*, pages 182–193. Springer, 2010.

[ADWL17] Tomer Ashur, Glenn De Witte, and Yunwen Liu. An automated tool for rotational-xor cryptanalysis of arx-based primitives. In *Proceedings of the 38th Symposium on Information Theory in the Benelux*, pages 59–66. Werkgemeenschap voor Informatie-en Communicatietheorie, 2017.

[AHJ+12] Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel sat solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 200–213. Springer, 2012.

[AHJP14] Gilles Audemard, Benoît Hoessen, Said Jabbour, and Cédric Piette. An effective distributed D&C approach for the satisfiability problem. In *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 183–187. IEEE, 2014.

[ALST16] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. An adaptive parallel sat solver. In *International Conference on Principles and Practice of Constraint Programming*, pages 30–48. Springer, 2016.

[AM13] Sk Subidh Ali and Debdeep Mukhopadhyay. Improved differential fault analysis of CLEFIA. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 60–70. IEEE, 2013.

[AMT13] Sk Subidh Ali, Debdeep Mukhopadhyay, and Michael Tunstall. Differential fault analysis of AES: towards reaching its limits. *Journal of Cryptographic Engineering*, 3(2):73–97, 2013.

[AS09a] Gilles Audemard and Laurent Simon. GLUCOSE: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.

[AS09b] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, volume 9, pages 399–404, 2009.

[AS12] Gilles Audemard and Laurent Simon. Refining Restarts Strategies for SAT and UNSAT. In *Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.

[AS14] Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 197–205. Springer, 2014.

[AS16] Gilles Audemard and Laurent Simon. Glucose and syrup in the sat'16. *SAT COMPETITION 2016*, pages 40–41, 2016.

[AS18] Gilles Audemard and Laurent Simon. Glucose and syrup: Nine years in the sat competitions. *Proc. of SAT Competition*, pages 24–25, 2018.

[Bar09] Gregory Bard. *Algebraic cryptanalysis*. Springer Science & Business Media, 2009.

[BBKN12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[BBR09] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-boolean constraints into CNF. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 181–194. Springer, 2009.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[BDF11] Charles Bouillaguet, Patrick Derbez, and Pierre-Alain Fouque. Automatic search of attacks on round-reduced AES and applications. In *Annual Cryptology Conference*, pages 169–187. Springer, 2011.

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[BECN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[BF15] Armin Biere and Andreas Fröhlich. Evaluating CDCL restart schemes. In *Pragmatics of SAT*, 2015.

[BGE+17] Jan Burchard, Mañl Gay, Ange-Salomé Messeng Ekossono, Jan Horáček, Bernd Becker, Tobias Schubert, Martin Kreuzer, and Ilia Polian. Autofault: towards automatic construction of algebraic fault attacks. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2017 Workshop on*, pages 65–72. IEEE, 2017.

[BGH+16] Curtis Bright, Vijay Ganesh, Albert Heinle, Ilias Kotsireas, Saeed Nejati, and Krzysztof Czarnecki. Mathcheck2: A SAT+ CAS verifier for combinatorial conjectures. In *International Workshop on Computer Algebra in Scientific Computing*, pages 117–133. Springer, 2016.

[BH+16] Tomáš Balyo, Marijn JH Heule, et al. Proceedings of sat competition 2016. 2016.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[Bie08a] Armin Biere. Adaptive restart Strategies for Conflict Driven SAT Solvers. In *Theory and Applications of Satisfiability Testing–SAT 2008*, pages 28–33. Springer, 2008.

[Bie08b] Armin Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[Bie10] Armin Biere. Lingeling, plingeling, Picosat and Precosat at SAT race 2010. *FMV Report Series Technical Report*, 10(1), 2010.

[Bie14] Armin Biere. Yet another local search solver and lingeling and friends entering the sat competition 2014. *SAT Competition*, 2014(2):65, 2014.

[Bie15] Armin Biere. Lingeling ayv. http://fmv.jku.at/lingeling/, 2015.

[Bie16] Armin Biere. Splatz, lingeling, plingeling, treengeling, yalsat entering the sat competition 2016. *Proc. of SAT Competition*, pages 44–45, 2016.

[Bie17] Armin Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2017. *Proceedings of SAT Competition*, pages 14–15, 2017.

[BKG18] Curtis Bright, Ilias Kotsireas, and Vijay Ganesh. A SAT+CAS method for enumerating Williamson matrices of even order. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2–7, 2018*, pages 6573–6580, 2018.

[BMZ05] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, 27(2):201–226, 2005.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. *Journal of CRYPTOLOGY*, 4(1):3–72, 1991.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference*, pages 513–525. Springer, 1997.

[BS18] Tomáš Balyo and Carsten Sinz. Parallel satisfiability. In *Handbook of Parallel Constraint Reasoning*, pages 3–29. Springer, 2018.

[BSS15] Tomáš Balyo, Peter Sanders, and Carsten Sinz. Hordesat: a massively parallel portfolio sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 156–172. Springer, 2015.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[CGP+08] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[Che14] Jingchao Chen. A bit-encoding phase selection strategy for satisfiability solvers. In *Theory and Applications of Models of Computation*, pages 158–167. Springer, 2014.

[CJW10] Nicolas T Courtois, Keith Jackson, and David Ware. Fault-algebraic attacks on inner rounds of DES. In *e-Smart'10 Proceedings: The Future of Digital Security Technologies*. Strategies Telecom and Multimedia, 2010.

[Cox58] David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):215–232, 1958.

[CP02] Nicolas T Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 267–287. Springer, 2002.

[CSH08] Geoffrey Chu, Peter J Stuckey, and Aaron Harwood. Pminisat: a parallelization of minisat 2.0. *SAT race*, 2008.

[DCR06] Christophe De Canniere and Christian Rechberger. Finding SHA-1 characteristics: general results and applications. In *Advances in Cryptology–ASIACRYPT 2006*, pages 1–20. Springer, 2006.

[DEM14] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. Analysis of SHA-512/224 and SHA-512/256. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 612–630. Springer, 2014.

[DKV07] Debapratim De, Abishek Kumarasubramanian, and Ramarathnam Venkatesan. Inversion Attacks on Secure Hash Functions Using SAT Solvers. *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 377–382, 2007.

[Dwo15] Morris J Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

[EMS14] Maria Eichlseder, Florian Mendel, and Martin Schläffer. Branching heuristics in differential collision search with applications to sha-512. In *International Workshop on Fast Software Encryption*, pages 473–488. Springer, 2014.

[ErH11] D Eastlake 3rd and Tony Hansen. US secure hash algorithms (SHA and SHA-based HMAC and HKDF). Technical report, 2011.

[ES] Niklas Eén and Niklas Sörensson. Minisat 2.2. http://minisat.se/.

[ES06] Niklas Eén and Niklas Sorensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[FF12] Hironori Fujii and Noriyuki Fujimoto. Gpu acceleration of bcp procedure for sat algorithms. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[FIP11] PUB FIPS. 180-4. *Federal Information Processing Standards Publication, Secure Hash*, 2011.

[FMM03] 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.

[FR12] Wieland Fischer and Christian A Reuter. Differential fault analysis on Grøstl. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on*, pages 44–54. IEEE, 2012.

[GD07] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification*, pages 519–531. Springer, 2007.

[GE03] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[GHJS10] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Diversification and intensification in parallel sat solving. In *International conference on principles and practice of constraint programming*, pages 252–265. Springer, 2010.

[GHR95] Raymond Greenlaw, H James Hoover, and Walter L Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press on Demand, 1995.

[GM11] Aurélien Garivier and Eric Moulines. *Algorithmic learning theory: 22nd International conference, ALT 2011, Espoo, Finland, October 5-7, 2011. Proceedings*, chapter On Upper-Confidence Bound Policies for Switching Bandit

Problems, pages 174–188. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[GMS94] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[GOS+12] 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.

[GS07] Matteo Gagliolo and Jürgen Schmidhuber. Learning Restart Strategies. In *IJCAI*, pages 792–797, 2007.

[HH11] Ludger Hemme and Lars Hoffmann. Differential fault analysis on the SHA-1 compression function. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pages 54–62. IEEE, 2011.

[HJB17] Marijn Heule, Matti Järvisalo, and Tomáš Balyo. Sat competition 2017. *SAT*, 2017.

[HJN10] Antti EJ Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning sat instances for distributed solving. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 372–386. Springer, 2010.

[HJS08] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2008.

[HJS16] Marijn Heule, Matti Järvisalo, and Martin Suda. SAT race benchmarks. http://satcompetition.org/sr2019benchmarks.zip, 2016.

[HJS18a] Marijn Heule, Matti Järvisalo, and Martin Suda. SAT competition benchmarks. http://sat2018.forsyte.tuwien.ac.at/benchmarks/, 2018.

[HJS+18b] Marijn JH Heule, Matti Juhani Järvisalo, Martin Suda, et al. Proceedings of sat competition 2018. 2018.

[HJS19] Marijn JH Heule, Matti Järvisalo, and Martin Suda. Sat competition 2018. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):133–154, 2019.

[HJT16] Marijn Heule, Matti Järvisalo, and Balyo Tomáš. SAT competition benchmarks. http://baldur.iti.kit.edu/sat-competition-2016/index.php?cat=downloads, 2016.

[HJT17] Marijn Heule, Matti Järvisalo, and Balyo Tomáš. SAT competition benchmarks. https://baldur.iti.kit.edu/sat-competition-2017/index.php?cat=benchmarks, 2017.

[HKB18] Marijn JH Heule, Oliver Kullmann, and Armin Biere. Cube-and-conquer for satisfiability. In *Handbook of Parallel Constraint Reasoning*, pages 31–59. Springer, 2018.

[HKWB11] Marijn JH Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65. Springer, 2011.

[HLMS14] Ronglin Hao, Bao Li, Bingke Ma, and Ling Song. Algebraic fault attack on the SHA-256 compression function. *International Journal of Research in Computer Science*, 4(2):1, 2014.

[HM12] Antti EJ Hyvärinen and Norbert Manthey. Designing scalable parallel sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 214–227. Springer, 2012.

[HMS+11] Steffen Hölldobler, Norbert Manthey, Julian Stecklina, Peter Steinke, et al. A short overview on modern parallel sat-solvers. In *Advanced Computer Science and Information System (ICACSIS), 2011 International Conference on*, pages 201–206. IEEE, 2011.

[HP16] Wei-Shou Hsu and Pascal Poupart. Online bayesian moment matching for topic modeling with unknown number of topics. In *Advances in Neural Information Processing Systems*, pages 4536–4544. Curran Associates, Inc., 2016.

[HR08] Michal Hojsík and Bohuslav Rudolf. Differential fault analysis of Trivium. In *International Workshop on Fast Software Encryption*, pages 158–172. Springer, 2008.

[HW09] Shai Haim and Toby Walsh. Restart Strategy Selection Using Machine Learning Techniques. In *Theory and Applications of Satisfiability Testing-SAT 2009*, pages 312–325. Springer, 2009.

[JCC+16] Priyank Jaini, Zhitang Chen, Pablo Carbajal, Edith Law, Laura Middleton, Kayla Regan, Mike Schaekermann, George Trimponias, James Tung, and Pascal Poupart. Online bayesian moment matching for topic modeling with unknown number of topics. In *International Conference on Learning Representations*, 2016.

[JK10] Philipp Jovanovic and Martin Kreuzer. Algebraic attacks using SAT-solvers. *Groups Complexity Cryptology*, 2(2):247–259, 2010.

[JL12] Kitae Jeong and Changhoon Lee. Differential fault analysis on block cipher LED-64. In *Future Information Technology, Application, and Service*, pages 747–755. Springer, 2012.

[JLSH13] Kitae Jeong, Yuseop Lee, Jaechul Sung, and Seokhie Hong. Security analysis of HMAC/NMAC by using fault injection. *Journal of Applied Mathematics*, 2013, 2013.

[JLU01] Bernard Jurkowiak, Chu Min Li, and Gil Utard. Parallelizing satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, 9:174–189, 2001.

[JP16] Priyank Jaini and Pascal Poupart. Online and distributed learning of gaussian mixture models by bayesian moment matching. *arXiv preprint arXiv:1609.05881*, 2016.

[JW90] Robert G Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.

[KJP14] Raghavan Kumar, Philipp Jovanovic, and Ilia Polian. Precise fault-injections using voltage and temperature manipulation for differential cryptanalysis. In *IOLTS*, pages 43–48. IEEE, 2014.

[KL06] Raihan H Kibria and You Li. Optimizing the initialization of dynamic decision heuristics in dpll sat solvers using genetic programming. In *European Conference on Genetic Programming*, pages 331–340. Springer, 2006.

[KLBS+16] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[KLT15] Stefan Kölbl, Gregor Leander, and Tyge Tiessen. Observations on the simon block cipher family. In *Annual Cryptology Conference*, pages 161–185. Springer, 2015.

[Knu86] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[Kre09] Martin Kreuzer. Algebraic attacks galore! *Groups–Complexity–Cryptology*, 1(2):231–259, 2009.

[KW14] Mark G. Karpovsky and Zhen Wang. Design of strongly secure communication and computation channels by nonlinear error detecting codes. *IEEE Trans. Computers*, 63(11):2716–2728, 2014.

[L+09] Tie-Yan Liu et al. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331, 2009.

[LAFW17] Pei Luo, Konstantinos Athanasiou, Yunsi Fei, and Thomas Wahl. Algebraic fault analysis of SHA-3. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 151–156. IEEE, 2017.

[Lam94] Leslie Lamport. *LATEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[LFBSK17] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. Painless: a framework for parallel sat solving. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 233–250. Springer, 2017.

[LFBSK19] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. Modular and efficient divide-and-conquer sat solver on top of the painless framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 135–151. Springer, 2019.

[LFZD16] Pei Luo, Yunsi Fei, Liwei Zhang, and A Adam Ding. Differential fault analysis of SHA3-224 and SHA3-256. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop on*, pages 4–15. IEEE, 2016.

[LGPC16a] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140. Springer International Publishing, 2016.

[LGPC16b] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140. Springer, 2016.

[LGZ+15] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. In *Haifa Verification Conference*, pages 225–241. Springer, 2015.

[LL12] Hsuan-Tien Lin and Ling Li. Reduction from cost-sensitive ordinal ranking to weighted binary classification. *Neural Computation*, 24(5):1329–1367, 2012.

[LLG09] Ruilin Li, Chao Li, and Chunye Gong. Differential fault analysis on SHACAL-1. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on*, pages 120–126. IEEE, 2009.

[LM93] Xucjia Lai and James L Massey. Hash Functions Based on Block Ciphers. In *Advances in Cryptology—EUROCRYPT'92*, pages 55–70. Springer, 1993.

[LM06] H Li and S Moore. Security evaluation at design time against optical fault injection attacks. *IEE Proceedings-Information Security*, 153(1):3–11, 2006.

[LNJVH14] Frédéric Lafitte, Jorge Nakahara Jr, and Dirk Van Heule. Applications of sat solvers in cryptanalysis: finding weak keys and preimages. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:1–25, 2014.

[LOG+17] Jia Hui Liang, Chanseok Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. Maple-comsps lrb vsids and maplecomsps chb vsids. *Proc. of SAT Competition*, pages 20–21, 2017.

[LOM+18] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for cdcl sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 94–110. Springer, 2018.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal Speedup of Las Vegas Algorithms. In *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*, pages 128–133. IEEE, 1993.

[LVP+17] Jia Hui Liang, Hari Govind VK, Pascal Poupart, Krzysztof Czarnecki, and Vijay Ganesh. An empirical study of branching heuristics through the lens of global learning rate. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 119–135. Springer, 2017.

[LW+02] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[Man16] Norbert Manthey. Towards next generation sequential and parallel sat solvers. *KI-Künstliche Intelligenz*, 30(3-4):339–342, 2016.

[Mas99] Fabio Massacci. Using Walk-SAT and Rel-SAT for Cryptographic Key Search. In *IJCAI*, volume 1999, pages 290–295, 1999.

[MBB11a] Mohamed Saied Emam Mohamed, Stanislav Bulygin, and Johannes Buchmann. Improved differential fault analysis of Trivium. *COSADE 2011*, pages 147–158, 2011.

[MBB11b] Mohamed Saied Emam Mohamed, Stanislav Bulygin, and Johannes A. Buchmann. Using SAT solving to improve differential fault analysis of trivium. In *ISA*, volume 200 of *Communications in Computer and Information Science*, pages 62–71. Springer, 2011.

[Mer89] Ralph C Merkle. One Way Hash Functions and DES. In *Advances in Cryptology—CRYPTO'89 Proceedings*, pages 428–446. Springer, 1989.

[MM00] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1-2):165–203, 2000.

[MMZ+01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[MNS11] Florian Mendel, Tomislav Nad, and Martin Schläffer. Finding sha-2 characteristics: searching through a minefield of contradictions. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 288–307. Springer, 2011.

[MNS13] Florian Mendel, Tomislav Nad, and Martin Schläffer. Improving local collisions: new attacks on reduced sha-256. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 262–278. Springer, 2013.

[Mor] Darius Morawiec. sklearn-porter. Transpile trained scikit-learn estimators to C, Java, JavaScript and others.

[MS13] Paweł Morawiecki and Marian Srebrny. A sat-based preimage analysis of reduced keccak hash functions. *Information Processing Letters*, 113(10-11):392–397, 2013.

[MSS99] João P Marques-Silva and Karem A Sakallah. GRASP: a search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[MVOV96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[MWGP11] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In *International Conference on Information Security and Cryptology*, pages 57–76. Springer, 2011.

[MZ06] Ilya Mironov and Lintao Zhang. Applications of sat solvers to cryptanalysis of hash functions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 102–115. Springer, 2006.

[NDT+20] Saeed Nejati, Haonan Duan, George Trimponias, Pascal Poupart, and Vijay Ganesh. Online bayesian moment matching based sat solver heuristics. *Submitted to ICML*, 2020.

[NG19] 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. IBM Corp., 2019.

[NHG+17] Saeed Nejati, Linag Jia Hui, Vijay Ganesh, Gebotys Catherine, and Czarnecki Krzysztof. Sha-1 preimage instances for sat. *Proc. of SAT Competition*, page 45, 2017.

[NHGG18] 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, Cham, 2018.

[NLFG20] Saeed Nejati, Ludovic Le Frioux, and Vijay Ganesh. A machine learning based splitting heuristic for divide-and-conquer solvers. *In preparation for CP*, 2020.

[NLG+17] 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, Cham, 2017.

[NNS+17] 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, Cham, 2017.

[Nos12] Vegard Nossum. SAT-based Preimage Attacks on SHA-1. 2012.

[Nos13] Vegard Nossum. Instance generator for encoding preimage, second-preimage, and collision attacks on sha-1. *Proceedings of the SAT competition*, pages 119–120, 2013.

[Oma16] Farheen Omar. *Online Bayesian Learning in Probabilistic Graphical Models using Moment Matching with Applications*. PhD thesis, University of Waterloo, 2016.

[Opt] Opturion. Opturion CPX 1.0.2. http://cpx.opturion.com/cpx.html. Accessed: 2018-03-30.

[OSG+16] Ilya Otpuschennikov, Alexander Semenov, Irina Gribanova, Oleg Zaikin, and Stepan Kochemazov. Encoding cryptographic functions to sat using transalg system. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pages 1594–1595. IOS Press, 2016.

[Pro16] Lukas Prokop. *Differential cryptanalysis with SAT solvers*. PhD thesis, University of Technology, Graz, 2016.

[PS15] Tobias Philipp and Peter Steinke. Pblib: a library for encoding pseudo-boolean constraints into CNF. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 9–16. Springer, 2015.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[Rin09] Jussi Rintanen. Planning and SAT. *Handbook of Satisfiability*, 185:483–504, 2009.

[RN18] Vadim Ryvchin and Alexander Nadel. Maple_lcm_dist_chronobt: Featuring chronological backtracking. *Proc. of SAT Competition*, pages 29–29, 2018.

[RP17] Francesco Regazzoni and Ilia Polian. Securing the hardware of cyber-physical systems. In *ASP-DAC*, pages 194–199. IEEE, 2017.

[RSV87] Richard L Rudell and Alberto Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, 1987.

[RZP16] Abdullah Rashwan, Han Zhao, and Pascal Poupart. Online and distributed bayesian moment matching for parameter learning in sum-product networks. In *Artificial Intelligence and Statistics*, pages 1469–1477, 2016.

[SB98] Richard S Sutton and Andrew G Barto. *Introduction to Reinforcement Learning*, volume 135. MIT Press Cambridge, 1998.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In *Annual International Cryptology Conference*, pages 570–596. Springer, Cham, 2017.

[SE05] Niklas Sorensson and Niklas Een. Minisat v1. 13-A SAT Solver with Conflict-Clause Minimization. *SAT*, 2005:53, 2005.

[SHA16] Ali Asgar Sohanghpurwala, Mohamed W Hassan, and Peter Athanas. Hardware accelerated sat solvers—a survey. *Journal of Parallel and Distributed Computing*, 2016.

[Sim14] Laurent Simon. Glucose 4.0. http://www.labri.fr/perso/lsimon/glucose/, 2014.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257. Springer, 2009.

[Soo15] Mate Soos. CryptoMiniSat 4.5.3. http://www.msoos.org/cryptominisat4/, 2015.

[Soo16] Mate Soos. The cryptominisat 5 set of solvers at sat competition 2016. *SAT COMPETITION 2016*, page 28, 2016.

[Soo18] Mate Soos. The cryptominisat 5.5 set of solvers at the sat competition 2018. *Proc. of SAT Competition*, pages 17–18, 2018.

[SST14] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In *International joint conference on automated reasoning*, pages 367–373. Springer, 2014.

[Ste] Stefan Kölbl. CryptoSMT: An easy to use tool for cryptanalysis of symmetric primitives. https://github.com/kste/cryptosmt.

[SW17] Nihar B Shah and Martin J Wainwright. Simple, robust and optimal ranking from pairwise comparisons. *The Journal of Machine Learning Research*, 18(1):7246–7283, 2017.

[SZ15] Alexander Semenov and Oleg Zaikin. Using monte carlo method for searching partitionings of hard variants of boolean satisfiability problem. In *International Conference on Parallel Computing Technologies*, pages 222–230. Springer, 2015.

[SZBP11] Alexander Semenov, Oleg Zaikin, Dmitry Bespalov, and Mikhail Posypkin. Parallel logical cryptanalysis of the generator a5/1 in bnb-grid system. In *International Conference on Parallel Computing Technologies*, pages 473–483. Springer, 2011.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *IFIP International Workshop on Information Security Theory and Practices*, pages 224–233. Springer, 2011.

[VDTHB12] Peter Van Der Tak, Marijn JH Heule, and Armin Biere. Concurrent cube-and-conquer. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 475–476. Springer, 2012.

[vWWM11] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. Practical optical fault injection on secure microcontrollers. In *FDTC*, pages 91–99. IEEE Computer Society, 2011.

[WSS16] Wenxi Wang, Harald Søndergaard, and Peter J Stuckey. A bit-vector solver with word-level propagation. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 374–391. Springer, 2016.

[XHHLB12] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 228–241. Springer, 2012.

[XLL+19] Fan Xiao, Chu-Min Li, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. A branching heuristic for sat solvers based on complete implication graphs. *Science China Information Sciences*, 62(7):72103, 2019.

[ZBH96] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4-6):543–560, 1996.

[ZGZ+13] Xinjie Zhao, Shize Guo, Fan Zhang, Zhijie Shi, Chujiao Ma, and Tao Wang. Improving and evaluating differential fault analysis on LED with algebraic techniques. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 41–51. IEEE, 2013.

[ZZG+13] Fan Zhang, Xinjie Zhao, Shize Guo, Tao Wang, and Zhijie Shi. Improved algebraic fault analysis: A case study on Piccolo and applications to other lightweight block ciphers. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 62–79. Springer, 2013.