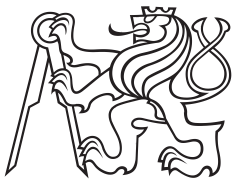
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering

Department of Computer Science

Performance analysis of a master-key system solver

Martin Hořeňovský

Open Informatics — Artificial Intelligence

horenmar@fel.cvut.cz

Jan 2018

Supervisor: Radomír Černoch, MSc.

Acknowledgement / Declaration

I would like to thank my supervisor for guidance given through the year I spent working on this, my friends for being (mostly) willing listeners when I ran into problems, and for their help with proofreading this text. I also want to thank my puppy for not trying to eat this thesis once it was printed out.

I hereby declare that I have done this work on my own and I declared all used sources according to "Metodický pokyn o dodržování etických principů při přípravě vysokoškolských závěřečných prací".

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

.....

Abstrakt / Abstract

Tato práce popisuje existující řešič systémů generálního klíče vyvinutý na katedře počítačů, FEL, ČVUT, který je založený na problému SAT. Práce poskytuje přehled, jak se systém generálního klíče převádí na problém SAT, jaké jsou primární faktory ovlivňující velikost výsledného problému SAT a jaké optimalizace směrované na zmenšení této velikosti byly v řešiči již implementovány.

Následně navrhuje několik změn způsobu, jakým je systém generálního klíče převáděn na SAT, a několik praktických optimalizací, například použití doménově závislých znalostí k poskytování rad řešiči SATu nebo změnu interní implementace MiniSatu za účelem jeho zrychlení. Dále je zkoumána rychlost rozdílných řešičů SATu na formulích generovaných katederním řešičem.

Všechny navržené změny jsou vyhodnoceny za použití reálných zakázek na výrobu systémů generálního klíče, které byly poskytnuty katederními průmyslovými partnery. Tyto změny také umožnily vyřešit dosud nevyřešený systém.

Na základě naměřených výsledků je nakonec navrženo několik dalších směrů pro navazující výzkum a práci.

Klíčová slova: Systém generálního klíče, mechanické klíče, mechanické zámky, splnitelnost logických formulí, SAT

Překlad titulu: Analýza výkonu řešiče systému generálního klíče a hlavních klíčů

This thesis describes an existing SAT-based master-key system solver developed at Department of Computer Science, FEE, CTU, providing an overview of how the solver converts a master-key system to SAT, what are the main factors affecting the size of the resulting SAT problem and what optimizations towards reducing the size have already been implemented.

It then proposes several changes to how the master-key system is converted to SAT, along with practical optimizations, such as using domain-specific knowledge to provide suggestions to the underlying SAT solver, or modifying MiniSat's internals to speed it up. The performance of different SAT solvers on computer-generated problems is also investigated.

All suggested changes are evaluated using a set of real-world inputs provided by the department's industrial partners resulting, among other things, in the modified algorithm finding a solution to a previously unsolved problem.

Finally, several areas of possible follow-up work are suggested based on the benchmarking results.

Keywords: Master-key system, mechanical keys, mechanical locks, boolean satisfiability programming, SAT

Contents /

1 Introduction 1
1.1 Mechanical locks and keys 1
1.2 Master-key systems 3
1.2.1 Lock-charts 3
1.2.2 Keyway profiles 4
1.3 Boolean Satisfiability problem (SAT) 4
1.4 CDCL SAT solvers 5
1.4.1 CDCL 6
1.4.2 Decision variables 6
1.5 Local search based SAT solvers .. 7
2 Description of a master-key problem 8
2.1 Customer provided lock-chart ... 8
2.2 Platform geometry 9
2.2.1 *General* constraints (*gecons*) 9
2.2.2 *Existential constraints* (*excons*) 10
2.2.3 *KeyDiff* constraints 10
2.2.4 *KeyDepthLockDepth* mappings 10
2.3 Solving a master-key system .. 11
2.3.1 Encoding physical properties 12
2.3.2 Encoding desired properties 13
2.3.3 Adding constraints 13
2.3.4 *KeyDepthLockDepth* mappings 15
2.3.5 Summary 16
2.4 Optimizations already present in the compiler 17
2.4.1 Simplifying lock definitions 17
2.4.2 Using implication in defining "stand-in" variables 18
3 Optimizing the conversion to SAT 19
3.1 Implication vs equivalence in variable definition 19
3.2 Different ways of formulating *KeyDiff* constraints 19
3.2.1 Generalization of the old scheme 20
3.2.2 "Direct" definition scheme 21
3.3 Reducing the number of decision variables 21
3.4 Reformulating profile positions 22
3.4.1 New profile formulation . 22
3.5 Summary 25
4 Optimizing compiler internals ... 27
4.1 Hinting assignments of variables 27
4.1.1 Applying key shape hint . 27
4.2 Memory consumption and SAT variable storage 28
4.3 Different SAT solvers 30
4.3.1 Reasons for the chosen versions 31
4.3.2 Unified SAT solver API . 31
4.4 Changing MiniSat's implementation of `lbool` 32
5 Results 35
5.1 Benchmarking setup 35
5.2 Settings and configurations 35
5.2.1 Configuration names 36
5.3 Inputs 36
5.3.1 Platform description 37
5.4 Evaluation methodology 38
5.5 Benchmark results 39
5.5.1 Closer look at the effect of individual settings 40
5.6 Evaluating changes to MiniSat implementation 42
6 Conclusion 43
References 45
A Specification 47
B Full results 49
C Glossary 53

Chapter 1
Introduction

Solving complex master-key systems is a surprisingly unexplored field of study given the real-world applications of results. The Department of Computer Science, FEE, CTU has developed a production ready master-key system solver based on work described in Radomír Černoch's doctoral thesis[1].

This thesis investigates the performance characteristics of the SAT-based master-key system solver, provides overview of the current state of the solver, including how the master-key system is converted to SAT, what factors affect the size of the resulting SAT problem and optimizations already implemented by the solver.

Further changes to how the existing solver converts master-key systems into SAT will be proposed, along with more practical optimizations such as using domain-specific knowledge to provide advice for the underlying SAT solver. The performance of newer SAT solvers on our specific problems will also be investigated.

All of these changes and optimizations will be evaluated using a set of non-trivial real-world inputs provided by our industrial partners and their effects on the department's solver run time will be reviewed.

The structure of this thesis is as follows: rest of this chapter provides an introduction into the working of mechanical locks and SAT solvers, chapter 2 provides a more formal description for the master-key problem and describes the current state of the department's master-key system solver. Chapters 3 and 4 explain the proposed changes, with chapter 3 focused on proposed changes in how the problem is converted to SAT and chapter 4 focused on the more practical optimizations. Chapter 5 explains the benchmarking methodology and analyzes the results. Finally, chapter 6 provides a conclusion to this thesis along with outlining a potential areas for further improvements.

To disambiguate our faculty's master-key system solver from the SAT solver it relies on, it shall be referred to as "SAT compiler", or "compiler", and "solver" shall be reserved for the underlying SAT solver hereafter.

1.1 Mechanical locks and keys

The underlying idea behind mechanical locks and keys is quite old, often dated to the ancient Egypt and sometimes even further back[2]. Although some places started using electronic cards and electric locks instead, there is still a large demand for manufacturing classical keys and locks, as both the mechanical and the electronic approach have their advantages. Some high-security implementations even use both independently, such as CLIQ[3].

The idea is that the lock contains a *tumbler*, a movable part that prevents the lock from opening. The tumbler is hard to move using lock-picking tools, but a correct key can

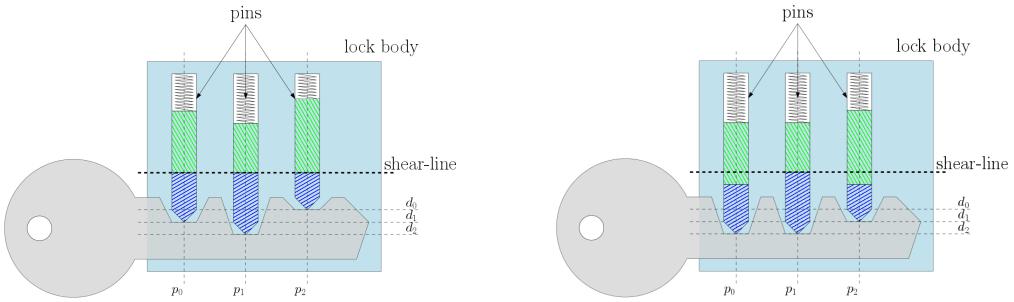


Figure 1.1. Pin tumbler lock schema. Blue and green parts of pins are disconnected by a cut. Left: Compatible key is inserted, cuts are aligned with the shear line. Right: Incompatible key is inserted, cuts are not aligned with the shear line.

move it away easily. This design can be implemented in many different ways; even for various European countries, we find that the key and lock designs differ significantly, e.g. a different core design for the tumbler is used in the Czech Republic and in the Scandinavian countries.

The most common and the most well-known lock type in the Czech Republic is the *pin tumbler lock*, so named because it contains spring-loaded pins that rest against the inserted key. Each pin has one or more horizontal cuts that correspond to cuttings on the key that opens the lock. For a lock to open, cuts of all its pins must align with the shear line. When a key does not open the lock, one or more of the pins are not aligned with the shear line. (Figure 1.1)

Modern pin tumbler locks require manufacturing equipment with low degree of mechanical tolerances, with the divide between the tumbler and the body of the lock often being smaller than a single millimeter. A view inside a modern pin tumbler lock is shown in figure 1.2.

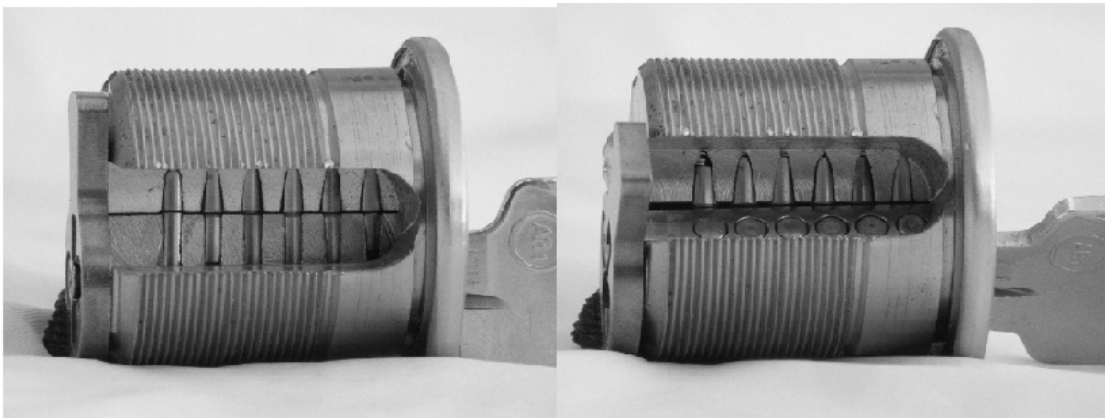


Figure 1.2. Lock for the *Mortise* door profile. *Left:* All pins are aligned with the shear line, when correct key is inserted into the lock. *Right:* The cylinder can be turned.
© 2003 by Matt Blaze

The specific type of locks, along with the number of pins, the number of possible cutting *depths*, manufacturing and security constraints, etc., is called a *platform*.

1.2 Master-key systems

A master-key system is a key-lock system where a single lock can be opened by multiple keys (and a single key may open multiple locks). These are commonly found in business buildings, where a typical person should only be able to access their own office, whereas some selected individuals should be able to open all doors on a floor and perhaps in the entire building. The most common schema used to specify which keys open which locks is called a *lock-chart*.

1.2.1 Lock-charts

Lock-charts are a way of encoding arbitrary *key opens lock* and *key does not open lock* (key is *blocked* in lock) constraints between a set of keys and a set of locks in a master-key system. The simplest way of visualizing a lock-chart is a full-sized table, such as the one in figure 1.3. This figure encodes a master-key system with 1 *general* key G, 2 *master* keys M_1 and M_2, 8 individual keys K_i, 8 *master keyed* locks L_i, and a *maison keyed* lock GL, for a total of 11 keys and 9 locks with non-trivial relationships.

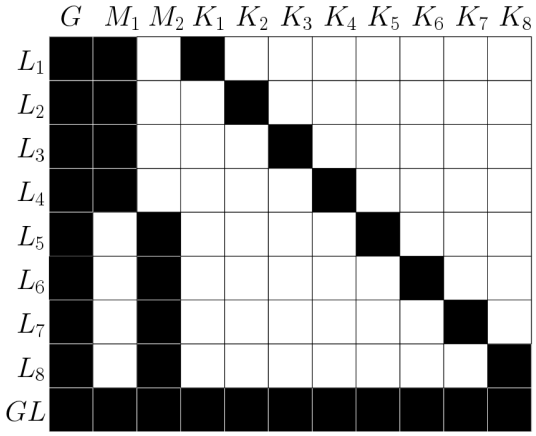


Figure 1.3. An example of a non-trivial lockchart

A lock-chart is considered solved when all keys and locks are assigned cutting depths, and the assignments conform to all *opens* and *blocked* constraints encoded in the lock-chart. Obviously this needs more information about the physical properties of keys and locks than are encoded in the lock-chart, at least the number of positions, and possible cutting depths at each position, are needed.

In the absence of other constraints, two kinds of lock-charts are proven to be solvable in polynomial time[1], a "diagonal" lock-chart, i.e. a lock-chart containing only master key and individual keys (as shown in figure 1.4), and a "key-to-differ" lock-chart, i.e. a lock-chart containing only individual keys (as shown in figure 1.5).

While a key-to-differ lock-chart is only rarely encountered in practice, diagonal lock-charts are quite common and are usually solved using the method known as *rotating constant method*[4].

The complexity of solving non-trivial kinds of lock-charts, such as the one in figure 1.3, is currently unknown. In the presence of specific kinds of constraints, solving any type of lock-chart is proven to be an NP-complete[1] problem.

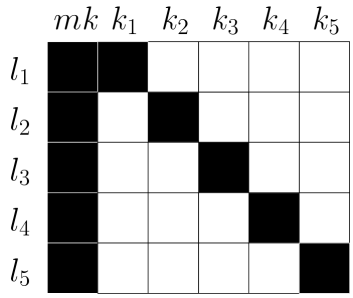


Figure 1.4. An example of a diagonal lock-chart with 5 locks

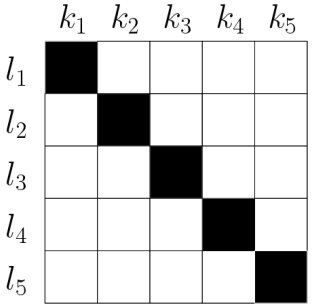


Figure 1.5. An example of a key-to-differ lock-chart with 5 locks

1.2.2 Keyway profiles

A *keyway* is the part of the lock that keys slide into. Generally, these can be cut in different ways, further disambiguating between different keys. Throughout this work, these cuts will be referred to as *profiles*[1]. Whereas a key with wrong cutting depths would be unable turn the tumbler inside the lock, a key with an incompatible keyway profile would not even enter the lock. For obvious reasons, each key and each lock can only have a single keyway profile.

The relations between profiles are encoded in a *profile map*. We can represent the map either as a directed graph, where a key with profile n is compatible with all locks with profiles that are reachable from node n, i.e. in figure 1.6, a key with profile 0 can open a lock with any profile, while a key with profile 1 can open locks with profile 1 or 3. An alternative way of representing profile maps can be seen in figure 1.7. This representation is very similar to a lock-chart, except that instead of showing which key opens which lock, it shows which key *profile* is compatible with which lock *profile*. Our compiler works with the latter representation.

An example of a real world profile hierarchy can be found in figure 1.8.

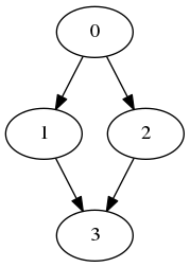


Figure 1.6. A small profile hierarchy as a directed graph

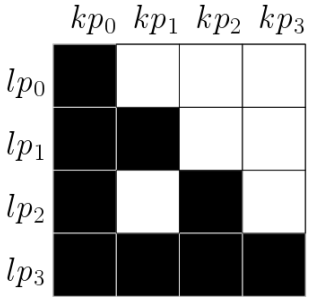


Figure 1.7. The same profile hierarchy as a profile map

1.3 Boolean Satisfiability problem (SAT)

Boolean satisfiability problem (SAT) is the problem of deciding whether a formula in boolean logic is satisfiable, i.e. whether there is at least one interpretation in which the

[1] Not to be confused with the external profile of a lock, such as the Mortise profile mentioned in figure 1.2.

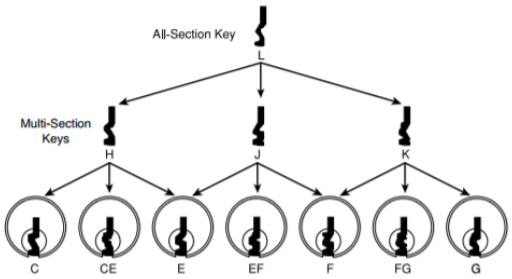


Figure 1.8. An example of a real-world profile hierarchy. © Allegion plc, 2014

formula evaluates to TRUE. SAT for formulas in conjuctive normal form (CNF) was the first problem proven to be NP-complete, by Cook[5], and can be used to prove NP-completeness of other problems, including solving master-key systems with non-trivial constraints.

SAT solvers are a very active area of research, with frequent competitions[1] between different SAT solvers. This means that it is easy to quickly find a high-quality implementation, and that these implementations accept a well-defined input format[2] for encoding CNF, making selecting and testing performance of different SAT solvers fairly easy.

Modern SAT solvers fall into one of 2 groups: *Conflict Driven Clause Learning* (CDCL) based solvers, or *local search* based solvers.

1.4 CDCL SAT solvers

CDCL based solvers are an evolution of DPLL[6] (Davis-Putnam-Logemann-Loveland) based solvers. DPLL algorithm is a complete and sound backtracking search algorithm. It works by selecting a variable to branch-on[3], sets its truth-value and propagates the truth-value into clauses. All clauses containing a positive literal (a literal that evaluates to true given the variable's truth value) of the propagated variable are deleted, and all occurrences of negative literals are removed from their respective clauses. Whenever an empty (i.e. unsatisfiable) clause is generated, the algorithm backtracks and tries different truth-value for variable(s). This is repeated until either all variables are set without generating an empty clause, or the algorithm has exhausted all possible truth-value assignments without succeeding.

What separates the DPLL from a naive backtracking algorithm is the usage of 2 simplification rules at each step, propagating *unit clauses* and eliminating variables that provably cannot affect the satisfiability of results. This is called *pure literal elimination* and happens whenever all occurrences of a variable have the same polarity. All clauses containing such variable can be trivially satisfied and thus do not provide further constraints on the satisfiability.

Unit clauses are clauses with only one unassigned literal. The only way to satisfy such clause is to assign the corresponding variable truth-value equal to the polarity of

[1] `http://satcompetition.org/`
[2] `http://www.satcompetition.org/2009/format-benchmarks2009.html`
[3] There are many different strategies and heuristics for the selection, but they are unimportant for this work.

the unassigned literal. This assignment can then be propagated in the same way as assignments done by the core backtracking loop of DPLL, including repeatedly propagating newly found unit clauses and removing pure literals. This process is called *unit propagation*.

1.4.1 CDCL

CDCL algorithm modifies how the DPLL algorithm backtracks. While DPLL backtracks chronologically, i.e. if both potential assignments of a variable lead to empty clauses, it then attempts to change assignment of the previously chosen variable, CDCL backtracks non-chronologically.

Specifically, when a CDCL based solver runs into a conflict (a variable would have to be assigned both TRUE and FALSE based on unit propagation[1]), it analyses the conflict, the clauses that caused it, and attempts to generate a *conflict clause*. This new clause is added to the portfolio of clauses, potentially[2] providing the solver with a clause that leads to conflicts quickly and allows the solver to backtrack (backjump) over multiple clauses.

CDCL solvers became widespread with MiniSat[7], an open source implementation of a state-of-the-art (as of 2003) SAT solver. MiniSat has shown that it is possible to implement a state-of-the-art SAT solver within a fairly minimal amount of code, and has been used as the basis for many newer experimental SAT solvers, such as Glucose[8]. Glucose innovates by keeping only certain kinds of learnt clauses and aggressively deleting other ones, thus accelerating unit propagation, which would otherwise be slowed down by superfluous clauses.

1.4.2 Decision variables

Naive conversion of a logic formula into CNF, using De Morgan laws and distributivity, can easily create a formula with exponential size. A typical example is converting a formula in a disjunctive normal form (DNF), e.g. $(x_0 \wedge x_1) \vee (x_2 \wedge x3) \vee (x_4 \vee x_5)$ is converted into six ternary clauses. In general case, a naive conversion of logical formula f consisting of N clauses in DNF would produce $\prod_{cl \in f} |cl|$ clauses of N literals.

To prevent this explosion in the size of a formula, a more efficient way of converting a formula into CNF can be used, i.e. Tseitin transformation[9]. These transformations introduce additional variables to the formula and the resulting CNF formula is not equivalent, but rather only equisatisfiable.

This means that in real-world problems, it is common for a problem to contain both "real" variables, that represent some truth about the actual problem, and "stand-in" variables, that were created in order to efficiently convert the original formulation into CNF. As an example, if we use SAT solver to find factors of a number, we would need to encode the working of binary multiplication circuits into CNF, but only the variables that represent inputs to the multiplication circuits need to be set as decision variables, all other variables will be inferred.

This can be exploited by the SAT solver when deciding which literal to branch on next, allowing it to branch on the "real" variables instead of the ones that stand-in for specific

[1] This is functionally equivalent to generating an empty clause, but can be detected somewhat earlier in practice.

[2] In practice, most of the generated clauses are useless and selecting helpful ones to keep is an active area of research.

assignment of other variables. Branching only on "real" variables does not necessarily have to be beneficial, because it limits the solver's exploration and thus the kind of clauses it can learn from conflicts. It also means that "non-decision" variables are set only if their value is inferred, e.g. by unit propagation. Soundness of a solution can be compromised in cases where the conflict would arise between non-decision variables that will not be inferred from values of decision-variables. If we mark decision variables x_i and non-decision variables y_i, then (1.1) shows an example of unsatisfiable CNF formula that the SAT solver will decide to be satisfiable.

$$(x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1) \wedge (\vee y_0 \vee y_1) \wedge (\neg y_0 \vee y_1) \wedge (y_0 \vee \neg y_1) \wedge (\neg y_0 \vee \neg y_1) \quad (1.1)$$

1.5 Local search based SAT solvers

Unlike CDCL based solvers, SAT solvers using local search are not necessarily complete, i.e. they might not find an existing solution, but can quickly provide solutions to large problems. In general, local-search based SAT solvers work by assigning a random truth-value to each variable and checking whether such assignment satisfies all clauses. If it does, the solver marks the formula as satisfiable and returns the current assignment. If the current assignment does not satisfy all clauses, then a variable needs to be flipped and the new assignment checked again. This process repeats until all clauses are satisfied, or some predetermined amount of time has elapsed, as termination is not guaranteed and the algorithms are not complete.

Two notable implementations of local-search based SAT solvers are WalkSat[10] and GSat[11]. GSat selects the variable to flip by determining how many clauses would be unsatisfied after the flip, and picking the variable that leads to the least number of unsatisfied clauses after flipping. To avoid being stuck in a local-minima, a random variable can also be selected with some probability. WalkSat picks an unsatisfied clause at random, and selects a variable that minimizes the number of newly unsatisfied clauses. Just as GSat, it can pick a random variable to flip in an attempt to escape local-minima.

Chapter 2
Description of a master-key problem

The input to our department's master-key solver consists of 2 parts:

- Customer provided lock-chart
- Description of platform geometry

The description of platform geometry can further be broken down into 3 parts:

- Description of cutting depths
- *Existential*, *General* and *KeyDiff* constraints
- *KeyDepthLockDepth* mappings

Various platform-specific constraints are translated into these in a preprocessing step that is not part of the solver itself.

2.1 Customer provided lock-chart

Each master-key problem must contain a lock-chart providing

- set K of all keys in the master-key system,
- set L of all locks in the master-key system,
- function $opens : K \to \{\ lock \mid lock\ is\ opened\ by\ key\ K\}$,
- function $openedBy : L \to \{\ key \mid key\ opens\ lock\ L\}$.

A lock-chart is always specific to one master-key system and thus is expected to change between problems.

If we take the example lock-chart from 1.3, then the above takes these concrete values

$$\begin{aligned} K &= \{0, 1, 2, \ldots, 10\} \\ L &= \{0, 1, 2, \ldots, 8\} \\ opens(0) &= \{0, 1, \ldots, 8\} \\ opens(1) &= \{0, 1, 2, 3, 8\} \\ opens(2) &= \{4, 5, \ldots, 8\} \\ opens(3) &= \{0, 8\} \\ &\ldots \end{aligned} \tag{2.1}$$

Notice that when describing master-key problem, they keys and locks are no longer named, only numbered.

2.2 Platform geometry

Platform geometry is a septuple consisting of:

- A *depth tuple*
- A set of *general* constraints
- A set of *existential* constraints
- A set of *KeyDiff* constraints
- Two functions to provide *KeyDepthLockDepth* mappings
- A set defining which positions in the depth tuple represent keyway profiles

For later reference, we also define set P as the set of all possible positions and set D as the set of all possible depths.

The *depth tuple* is a p-tuple of natural numbers $(n_0, n_1, ..., n_{p-1})$, where n_i is the maximal cutting depth at position i. The set of possible cutting depths for position i is assumed to be a set of natural numbers, $\{0, 1, 2, ..., n_i\}$. If any of these cutting depths needs to be forbidden, it can be done using a *general* constraint.

This means that a depth tuple (2, 2, 2, 2) specifies maximal cutting depths for 4 positions, each of which can be cut to depth 0, 1 or 2. It also defines set P as $\{0, 1, 2, 3\}$ and set D as $\{0, 1, 2\}$.

If the given platform contains keyway profiles, they are transformed into new cutting position(s), called profile position(s), and the profile maps (similar to lock-chart, but for profiles) are encoded into *KeyDepthLockDepth* functions, as explained later. This often leads to an "un-even" geometry, with the last couple of positions having significantly different maximal cutting depth from the other ones. As an example that illustrates the typical size of the problem, a real-world platform that uses profile positions may have a depth tuple of (8, 8, 8, 8, 8, 8, 60), with the last position being the profile position.

Geometry description can be expected to remain the same across many inputs, because each platform is designed to service a large number of customer requests.

2.2.1 *General* constraints (*gecons*)

In the context of this thesis, a *gecon* is the same as defined in Radomír Černoch's doctoral thesis[1], but using a different, more compact, representation.

One *gecon* is an ordered tuple of (Position, Depth) pairs that are forbidden from appearing in a cutting. A key whose cutting does not match all of the *gecon*'s constituent (Position, Depth) pairs is allowed. As an example, *gecon* ((0, 2), (1, 3), (2, 4)) forbids a key cutting of (2, 3, 4), but allows any of (1, 3, 4), (2, 2, 4), (2, 3, 3).

An example of constraint that gets preprocessed into a set of *gecon* constraints is the so-called *jump*. A *jump* of 3 means that cuttings at two neighbouring positions in a key cannot have depth difference larger than 3. For example, assuming a very simple depth tuple of (5, 5), the result of such preprocessing is a set of 6 *gecons* shown in table 2.1.

$((0,0),(1,4))$	$((0,0),(1,5))$	$((0,1),(1,5))$
$((0,4),(1,0))$	$((0,5),(1,0))$	$((0,5),(1,1))$

Table 2.1. Example results of processing *jump* constraint into *gecons*

In the general case, a *jump* is translated into a polynomial number of *gecons* as seen in (2.2)[1]

$$(2 \cdot \left(\sum_{i=1}^{j} d - j - i \right) + (d - 2j) \cdot (d - 2j - 1)) \cdot (p - 1) \tag{2.2}$$

In the solver itself, a *gecon* can either apply to any key/lock, or a specific key/lock. This can be used for e.g. disabling a cutting for the master key, but leaving it enabled for the individual keys.

2.2.2 *Existential constraints (excons)*

An *existential constraint* (*excon*) serves to constrain cutting depths occurring in a key on any position. An *excon* is an n-tuple of cutting depths, at least one of which has to be present in the key for it to satisfy the constraint. For example, *excon* $(1, 3)$ is satisfied by all of these cuttings: (1, 1, 1), (2, 2, 3) and (1, 3, 1), but is not satisfied by cutting (0, 2, 4), because neither 1 nor 3 is contained in the cutting.

One of the requirements that are easily convertible into *excons* is a requirement on the minimum difference between the deepest and the shallowest cutting depth in a key. This is done to prevent keys that are too "straight" and would cause locks to open with all pins in a single plane.

As an example, if possible cutting depths of keys are 1-6 and the manufacturer requirement is that the difference is at least 2, 5 *excons* are generated: (3, 4, 5, 6), (1, 4, 5, 6), (1, 2, 5, 6), (1, 2, 3, 6), (1, 2, 3, 4).

2.2.3 *KeyDiff* constraints

KeyDiff constraint is a triplet (k_1, k_2, n), that defines the amount of differentiation between keys k_1 and k_2, to n. Specifically, it constraints how many positions can the cutting of two keys differ at, e.g. (1, 1, 1, 1, 2) and (1, 1, 1, 99, 2) would have a *KeyDiff* of 1, as they differ at only one position and the size of the individual difference does not matter.

There are two kinds of *KeyDiff* constraints, *MinKeyDiff* and *MaxKeyDiff*. A *MinKeyDiff* sets the minimum difference[2] and *MaxKeyDiff* sets the maximum difference between a pair of keys. An alternative way of looking at things is that *MaxKeyDiff* of m sets the minimum number of positions that have to match between a pair of keys to $|P| - m$.

2.2.4 *KeyDepthLockDepth* mappings

The *KeyDepthLockDepth* mappings are 2 functions

- $blocked : P \times D \to \{\{depths\}, \{depths\}, ...\}$,
- $implies : P \times D \to \{\{depths\}, \{depths\}, ...\}$

[1] There is a mistake in Radomír Černoch's thesis stating that it is $\frac{1}{2} * jump * (jump + 1) * (p - 1)$, but that is an obvious oversight, as it would mean that the amount of generated constraints does not depend on the platform geometry depths.

[2] Note that the problem structure implies that the difference between two keys with different *opens* has a *MinKeyDiff* of at least 1.

that map key's cutting depth d at position p onto a set of sets of cutting depths for the same position in a lock.

One possible way of looking at *KeyDepthLockDepth* mappings is that they return a set of corresponding "superpositions" in lock to a (position, depth) pair in key, where the superpositions are arbitrary combinations of depths. The meaning of the *implies* mapping is that for the key to open lock at given position, at least one of the superpositions must be cut inside the lock. A superposition is considered cut when all of its constituent depths are cut.

Because blocking is, by definition, the negation of opening, the meaning of the *blocking* mapping is that none of the superpositions can be cut inside the lock for a key to be blocked in the lock. A superposition is not cut when at least one of its constituent depths is not cut.

KeyDepthLockDepth mappings often serve to encode profiles at a position. As an example, the profile map in 1.6 would be encoded into *KeyDepthLockDepth* mappings as seen in (2.3).

$$\begin{aligned}
implies(p,0) = blocked(p,0) &= \{\{0\},\{1\},\{2\},\{3\}\} \\
implies(p,1) = blocked(p,1) &= \{\quad\{1\},\quad\{3\}\} \\
implies(p,2) = blocked(p,2) &= \{\quad\quad\{2\},\{3\}\} \\
implies(p,3) = blocked(p,3) &= \{\quad\quad\quad\{3\}\}
\end{aligned} \tag{2.3}$$

In the basic physical model of keys and locks, each (position, depth) pair cut in a key requires the exact same (position, depth) pair to be cut in a lock for it to open. This is equivalent with the semantics of defining both *KeyDepthLockDepth* mappings to return $\{\{d\}\}$ for given (position, depth) pair. This means that, for the sake of simplicity, it suffices to provide *KeyDepthLockDepth* mappings only when they are non-trivial. With that in mind, we define "Profile positions" as the set of all positions for which an *implies* mapping has been provided.[1]

2.3 Solving a master-key system

To solve a master-key system, the compiler converts it into a CNF that can be given to a SAT solver. To simplify potential cross-referencing, I will be using the same notation as Radomír Černoch's thesis[1], where possible, and attempt to keep new notation similar.

Because the original problem domain is naturally discrete, converting the Master-key specification into SAT can be done in a fairly straightforward manner, though different formulations might have different advantages and disadvantages, which we discuss in later chapters.

When converting a master-key system into SAT, we can roughly split the process into 3 parts:

- Encoding the physical properties

[1] Theoretically an input could require *implies* mapping for positions that are not transformed profiles, but we have not come across such an input yet.

- Encoding the desired properties
- Encoding the constraints on the solution

The physical properties of a system define locks and keys and their basic properties, such as that a key can be cut to only one depth at given position. The desired properties define relationships between locks and keys, e.g. the fact that key 1 should open every lock. Finally, the constraints can forbid certain solutions, e.g. to prevent repeated generation of an existing system, or to avoid systems that could not be manufactured because of tolerances involved in manufacturing the system.

The distinction between these is not always completely clear-cut.

2.3.1 Encoding physical properties

To define physical properties of keys, a total of $|K| * p * d$ variables has to be created, one for each depth, each position and each key. A variable for position p, depth d and key k is denoted as $key^k_{p,d}$.

To encode that each position in a key must have exactly 1 cutting depth, we create a number of clauses. First, we force at least one cutting depth for each position by adding $|K| * p$ clauses:

$$\bigvee_{d \in D} key^k_{p,d} \tag{2.4}$$

Then, we enforce at most 1 cutting depth for each position by adding $|K| * p * \frac{d(d-1)}{2}$ clauses:

$$\bigwedge_{d_1,d_2 \in D,\ d_1<d_2} key^k_{p,d_1} \Rightarrow \neg key^k_{p,d_2} \tag{2.5}$$

or, as CNF:

$$\bigwedge_{d_1,d_2 \in D,\ d_1<d_2} \neg key^k_{p,d_1} \vee \neg key^k_{p,d_2} \tag{2.6}$$

To define physical properties of locks, a total of $|L| * p * d$ variables needs to be created. A variable for position p, depth d and lock l is denoted as $lock^l_{p,d}$. Unlike keys, locks can have an arbitrary number of cuttings per position, so it is enough to enforce[1] at least 1 cutting depth per position, by adding $|L| * p$ clauses:

$$\bigvee_{d \in D} lock^l_{p,d} \tag{2.7}$$

The above is not sufficient when a platform differentiates between key profiles, because at profile positions, locks can also have at most 1 cutting depth. In this case, we have to add further clauses per each profile position and lock:

$$\bigwedge_{d_1,d_2 \in D, d_1<d_2} \neg lock^l_{p,d_1} \vee \neg lock^l_{p,d_2} \tag{2.8}$$

[1] Note that under certain circumstances, even these clauses can be omitted. This will be further explained later on.

2.3.2 Encoding desired properties

Essentially, there are 2 desired properties in a master-key system:

- Keys open certain locks
- Keys do not open (are *blocked in*) other locks

Because a key opens a lock when the cutting of the key allows for the pin in the lock to clear the shear line, translation to our SAT model is simple:

$$\bigwedge_{p\in P,d\in D} key^k_{p,d} \Rightarrow lock^l_{p,d} \tag{2.9}$$

or, in CNF:

$$\bigwedge_{p\in P,d\in D} \neg key^k_{p,d} \vee lock^l_{p,d} \tag{2.10}$$

This definition allows for cutting depths in locks that are not matched to any key, which is undesirable as each extra cutting increases manufacturing costs and decreases lock security. However, once a solution is found, superfluous cuts in locks can be removed in polynomial time by checking all keys that open given lock for their cutting depths at given position.

For a key to be *blocked* in a lock, their cutting must differ in at least one position. This can also be written down in a straightforward manner, as seen in (2.11).

$$\bigvee_{p\in P,d\in D} (key^k_{p,d} \wedge \neg lock^l_{p,d}) \tag{2.11}$$

However, a straightforward conversion of this formulation into CNF results in 2^P clauses of length P. Instead, let us declare a new variable, $blocking^{k,l}_{p,d}$, as seen in (2.12),

$$key^k_{p,d} \wedge \neg lock^l_{p,d} \iff block^{k,l}_{p,d} \tag{2.12}$$

for each blocked key-lock pair and each position. Then we can define one blocking clause for each key-lock pair as seen in (2.13).

$$\bigvee_{p\in P,d\in D} block^{k,l}_{p,d} \tag{2.13}$$

Some key manufacturers prefer a key to be blocked in lock via multiple positions, for added security and increased manufacturing tolerance. This can be done using the same idea that is described in section 3.2.1 for *MinKeyDiff* constraints, but replacing *comp_total* variables with *blocking* variables for given key and lock pair.

2.3.3 Adding constraints

There are three ways to introduce a constraint on the solution into the solver:

- *gecons*
- *excons*
- *KeyDiff* constraints

A *gecon* can be translated in a straightforward manner, whereas *excons* and *KeyDiff* constraints require introducing additional variables.

A *gecon* can apply to either all keys, all locks, single key or a single lock. For simplicity, we can consider *gecon* that applies to all keys the same as $|K|$ *gecons* one for each key, and similarly a *gecon* for all locks is equal with $|L|$ constraints, one for each lock. This means that we only need to translate the latter two kinds of *gecons* into SAT.

Each *gecon* creates a single clause. Given a *gecon* for key k, it is as follows:

$$\bigvee_{(p,d)\in gecon} \neg key^k_{p,d} \tag{2.14}$$

As an example, converting one of the *gecons* from section 2.2.1, specifically $((0,0),(1,5))$, for key 3 generates this clause:

$$\neg key^3_{0,0} \vee \neg key^3_{1,5} \tag{2.15}$$

A *gecon* for a lock is converted in the same manner, with the difference that it applies to a set of *lock* instead:

$$\bigvee_{(p,d)\in gecon} \neg lock^k_{p,d} \tag{2.16}$$

The straightforward way of converting *excon* to SAT is to create clause over *key* variables for all positions and all depths contained in the *excon*, as shown in figure (2.17).

$$\bigvee_{p\in P} \bigvee_{d\in excon} key^k_{p,d} \tag{2.17}$$

This leads to $|K|$ clauses of length $|excon|\cdot|P|$ per each *excon* constraint. However, our compiler instead uses a different translation. First, we define a new variable $shape^k_d$, that is true if key k is cut to depth d in any position and false otherwise, as seen in figure (2.18). Using these variables, we can translate *excon* constraint as seen in figure (2.19).

$$shape^k_d \iff \bigvee_{p\in P} key^k_{p,d} \tag{2.18}$$

$$\bigvee_{d\in excon} shape^k_d \tag{2.19}$$

This approach leads to approximately[1] $|K|\cdot|D|$ new variables, $|K|\cdot|D|$ clauses of length $|P|$ and $|K|\cdot|D|\cdot|P|$ binary clauses to define these variables, and $|K|$ clauses of length $|excon|$ to translate each *excon*.

At first glance, using the second translation of *excons* might appear inefficient, but there are a couple factors in its favour:

- The *shape* variables and their assorted clauses are defined once and can be reused for each *excons* constraint.

[1] The compiler defines variables lazily, so if a *shape* variable would not be required, it is not defined.

- Handling of binary clauses in modern SAT solvers is very efficient.

The way *shape* variables are defined can also be optimized further, as will be covered in section 2.4.

To define *KeyDiff*, more helper variables need to be defined, namely $comp_{p,d}^{k_1,k_2}$, as shown in (2.20) and $comp\_total_{p}^{k_1,k_2}$, as shown in (2.21).

$$comp_{p,d}^{k_1,k_2} \iff (key_{p,d}^{k_1} \iff key_{p,d}^{k_2}) \tag{2.20}$$

$$comp\_total_{p}^{k_1,k_2} \iff \bigvee_{i=0}^{d} \neg comp_{p,i}^{k_1,k_2} \tag{2.21}$$

Using these definitions, $comp\_total_{p}^{k_1,k_2}$ is true when keys k_1 and k_2 have different depth cut on position p. This works because $comp_{p,d}^{k_1,k_2}$ is true if keys k_1 and k_2 either both have, or both do not have cut in position p at depth d. Note that either all $comp^{k_1,k_2}$ variables will be true, or two will be false, because key can only have a single cutting depth at a given position.

In total, for each pair of keys, $p+p\cdot d$ variables will be defined, using $p\cdot d$ binary clauses, $4\cdot p\cdot d$ ternary clauses and p clauses of length d.

Using these variables a *MinKeyDiff* (k_1, k_2, 1) is converted into a disjunction over all *comp_total* variables for the given key pair, as seen in (2.22).

$$\bigvee_{p\in P} comp\_total_{p}^{k_1,k_2} \tag{2.22}$$

A *MinKeyDiff* (k_1, k_2, 2) is converted by reusing conversion for *MinKeyDiff* with difference of 1, and adding a set of implications from (2.23).

$$\bigwedge_{p_1\in P} (comp\_total_{p_1}^{k_1,k_2} \Rightarrow \bigvee_{p_2\in P, p_1\neq p_2} comp\_total_{p_2}^{k_1,k_2}) \tag{2.23}$$

This works because the conversion from (2.22) forces at least 1 position to be different. Then the set of implications from (2.23) mean that if a position differs, then at least 1 more, different, position has to differ between the two keys.

Note that the original compiler only supports *MinKeyDiff* with a difference of 1 or 2; the support for other *KeyDiff* constraints is one of the outcomes of my work. A generalized schema for both *MinKeyDiff* and *MaxKeyDiff* constraints with an arbitrary difference will be provided in section 3.2.

2.3.4 *KeyDepthLockDepth* mappings

KeyDepthLockDepth mappings change how *blocked-in* and *opens* relations between keys and locks are converted to SAT. To simplify notations in this section, let us define $\prod$ as Cartesian product and shorten *implies* function as i and *blocked* as b. Let us also define P_i as $\prod_{D\in i(p,d)} |D|$ and P_b as $\prod_{D\in b(p,d)} |D|$, meaning that P_i is the number of

tuples generated as cartesian product over sets in *implies* and P_b is the same, but for *blocking*.

The *KeyDepthLockDepth* mappings do not add clauses directly, but rather change how a key *opens* lock and key *blocked in* lock relations are encoded. For *implies*, the formulation in (2.9) is replaced with set of clauses generated in (2.24).

$$\bigwedge_{(d_0,d_1,\ldots,d_n)\in\prod_{D\in i(p,d)} D} key^k_{p,d} \Rightarrow (lock^l_{p,d_0} \vee lock^l_{p,d_1} \vee \ldots \vee lock^l_{p,d_n}) \tag{2.24}$$

This means that P_i clauses with size $|i(p,d)|+1$ are created for *implies* when *KeyDepthLockDepth* mappings are used. We can also see that the formulation in (2.9) is just a special case of *KeyDepthLockDepth* mapping, that returns a single singular set.

For *blocked* mapping, the changes are done to how the *blocking* variables are defined. The new formulation can be derived by starting with negating (2.24). By applying De Morgan's laws, we get (2.25). This is a DNF, which is efficiently converted to CNF by substituting the inner conjunction for a *blocking* variables, as shown in (2.26).

$$\bigvee_{(d_0,d_1,\ldots,d_n)\in\Pi_{D\in b(p,d)} D} (key^k_{p,d} \wedge lock^l_{p,d_0} \wedge \ldots \wedge lock^l_{p,d_n}) \tag{2.25}$$

$$\forall_{(d_0,\ldots,d_n)\in\prod_{D\in b(p,d)}} : block^{k,l,(d_0,\ldots,d_n)}_{p,d} \iff (key^k_{p,d} \wedge \neg lock^l_{p,d_0} \wedge \ldots \wedge \neg lock^l_{p,d_n}) \tag{2.26}$$

This means that the formulation in (2.12) is replaced by the formulation in (2.26) for positions and depths that have a *KeyDepthLockDepth* mapping.

This means that P_b *blocking* variables, $P_b \cdot (|b(p,d)|+1)$ binary clauses and P_b clauses of length $|b(p,d)|+2$ are created per each position, depth per key, lock pair that is blocked. Once again, the formulation in (2.12) can be seen as special case of *blocking KeyDepthLockDepth* mapping returning single singular set.

Finally, for positions that originated as keyway profiles we know that the inner sets returned from *KeyDepthLockDepth* mappings are always singular. This means that $P_i = P_b = 1$ for profile positions.

2.3.5 Summary

Before trying to summarize the size of resulting CNF input to a SAT solver, we need to define some auxiliary sets and terms:

- p is the number of positions (the size of *depth tuple*),
- nd is the number of all possible cutting depths (or $\Sigma_{i=0}^{p}(d_i+1)$),
- no is the number of all (key, lock) pairs that open,
- nb is the number of all (key, lock) pairs that block,
- kp is the number of all different key pairs (or $\frac{|K|\cdot|K-1|}{2}$),

	variables	clauses	
		number	size
keys	$\|K\| \cdot nd$	$\|K\| \cdot p \cdot \frac{nd(nd-1)}{2}$	2
		$\|K\| \cdot p$	d_i
locks	$\|L\| \cdot nd$	$\|L\| \cdot p$	d_i
converted profiles		$\|L\| \cdot p \cdot \frac{nd(nd-1)}{2}$	2
opening	—	$no \cdot nd$	2
implies mapping	—	P_i	$\|i(p,d)\| + 1$
blocking	$nb \cdot nd$	$2 \cdot nb \cdot nd$	2
		$nb \cdot nd$	3
		nb	nd
blocked mapping	$nb \cdot nd \cdot P_b(p,d)$	$nb \cdot nd \cdot P_b(p,d) \cdot (\|b(p,d)\| + 1)$	2
		$nb \cdot nd \cdot P_b(p,d)$	$\|b(p,d)\| + 2$
gecons	—	$\|gecons\| \cdot \|K\|^1$	$\leq p$
Excons	$\|K\| \cdot d$	$\|Excons\| \cdot \|K\|$	$\leq d$
		$\|K\| \cdot d$	$1 + p$
		$\|K\| \cdot d \cdot (p+1)$	2
comp variables	$kp \cdot p \cdot d$	$4 \cdot kp \cdot p \cdot d$	3
comp_total variables	$kp \cdot p$	$kp \cdot p \cdot d$	2
		$kp \cdot p$	d
MinKeyDiff 1	—	$\|MinKeyDiff\|$	p
MinKeyDiff 2	—	$\|MinKeyDiff\| \cdot (p+1)$	p

Table 2.2. Summary of the size of CNF generated using simple conversion

Using these terms we can calculate the effect of each part of the conversion on the size of the resulting CNF, as shown in table 2.2. It is important to note that for large lock-charts we expect $nb \gg no$ to hold. Furthermore, given that $no + nb = |K| \cdot |L|$ by definition, we also expect nb to approach $|K| \cdot |L|$.

This means that for sufficiently large systems, the two dominating factors for CNF size is the implementation of blocking between keys and locks, and defining *comp* and *comp_total* variables for use in implementation of *KeyDiff* constraints.

2.4 Optimizations already present in the compiler

This section gives an overview of existing optimizations in the compiler that make the resulting CNF smaller, as compared to the naive translation that has been the focus of this chapter so far.

2.4.1 Simplifying lock definitions

In (2.7) we define clauses ensuring that each lock has at least one cutting depth at each position. However, these can be skipped for each lock that is opened by at least one key, as the cutting depths in key will force at least one cutting depth in the lock because of (2.9), otherwise the lock could not be opened by the key.

This saves $|L| \cdot p$ clauses of length d_i.

2.4.2 Using implication in defining "stand-in" variables

The *blocking* variables in section 2.3.2 are defined as an equivalence, because that is their formal meaning. This leads to 3 clauses per variable, but 2 binary clauses could be saved by using implication instead, as shown in (2.27). Logical propagation in this formulation is different, but the resulting CNF is still equisatisfiable, at least for pure SAT model.

Providing a full proof would take a considerable amount of paper, but consider this quick sketch of one of the implications: Assume that we have TRUE/FALSE assignments for all *key* and *lock* variables. If an equivalence-based model is satisfied by such assignment, then the implication-based model is also satisfied by such assignment. This can be seen from the fact that the implication-based CNF is a subset of the equivalence-based CNF, and if a CNF is satisfied, then all of its subsets are also satisfied.

$$block_{p,d}^{k,l} \Rightarrow key_{p,d}^{k} \wedge \neg lock_{p,d}^{l} \tag{2.27}$$

This saves $nb \cdot p \cdot nd$ binary clauses.

Just like *blocking* variables, the *shape* variables in section 2.3.3 are defined as an equivalence, because that is their formal meaning, but can be instead defined using an implication, as shown in (2.28). This saves p binary clauses per *shape* variable, while still being equisatisfiable in a pure SAT model.

$$shape_{d}^{k} \Rightarrow \bigvee_{p \in P} key_{p,d}^{k} \tag{2.28}$$

Chapter 3
Optimizing the conversion to SAT

This chapter goes over various details of the conversion to SAT presented in the previous chapter and optimization opportunities they present. Namely, the opportunities consist of

- switching between defining *blocking* variables using implication or equivalence,
- switching between defining *shape* variables using implication or equivalence,
- different ways of defining *opens* and *blocks* for profile positions,
- using the concept of decision variables to guide the problem space exploration performed by the SAT solver,
- different ways of implementing *KeyDiff* constraints.

The chapter does not concern itself with more practical concerns, such as choice of the SAT solver, efficient in-memory representation of the input, nor speed-ups gained from improving the implementation of either our compiler, or the underlying SAT solver. These concerns are left to chapter 4.

3.1 Implication vs equivalence in variable definition

Although it was mentioned in section 2.4 as an optimization, the practical effect of defining *blocking* and *shape* variables as an implication, instead of as an equivalence, is not so clear-cut.

In practice, a larger number of clauses can allow the SAT solver to find conflicts quicker, or derive better learnt clauses. Having more short clauses also speeds up unit propagation, especially when the clauses are binary. Because removing the second implication in definitions of *blocking* and *shape* variables removes only binary clauses, it might be beneficial to keep them in the resulting CNF.

As both *blocking* and *shape* variables can use either model independently, there are 4 different configurations to test:

- both *blocking* and *shape* variables are defined using equivalences,
- *blocking* variables are defined using equivalences, *shape* variables using implications,
- *blocking* variables are defined using implications, *shape* variables using equivalences,
- both *blocking* and *shape* variables are defined using implications.

3.2 Different ways of formulating *KeyDiff* constraints

As mentioned in 2.3.3, the compiler used to only implement converting the *MinKeyDiff* variant of *KeyDiff* constraint, with possible differences limited to 1 or 2. This section

describes a generalized scheme for arbitrary differences, a different scheme for arbitrary differences and how they apply to *MaxKeyDiff* constraints. The SAT compiler uses the same scheme for converting both *MinKeyDiff* and *MaxKeyDiff* constraints, leading to 2 configurations to test.

3.2.1 Generalization of the old scheme

The original scheme builds up clauses for difference 2, by reusing the clause for difference 1 to force at least 1 position to differ, and then adding an implication for each position saying that if this position differs, then at least one other position has to differ. With some extra effort, this recursive scheme can be generalized to work for an arbitrary difference between two keys.

To enforce a *MinKeyDiff* (k_1, k_2, n), we need to enforce a *MinKeyDiff*(k_1, k_2, $n-1$) and then create a new set of implications, as seen in (3.1).

$$\bigwedge_{sel \in selectK(P,n-1)} \left(\bigwedge_{p_i \in sel} comp\_total_{p_i}^{k_1,k_2} \right) \Rightarrow \left(\bigvee_{p_j \notin sel} comp\_total_{p_j}^{k_1,k_2} \right) \tag{3.1}$$

where *selectK(S, K)* is a function that returns all K-combinations of items from set S. For *MinKeyDiff* (k_1, k_2, 1), a disjunction of all *comp_total* variables is generated, as shown in (2.22).

Enforcing a *MaxKeyDiff* (k_1, k_2, n) works analogously, but with the logic inverted. The basic idea is that if two keys are to differ in at most n positions, then $|P| - n$ positions must be the same. Using this insight, we can enforce a *MaxKeyDiff* (k_1, k_2, n) by enforcing *MaxKeyDiff* (k_1, k_2, $n+1$) and then a new set of implications is created, as shown in (3.2).

$$\bigwedge_{sel \in selectK(P,|P|-n-1)} \left(\bigwedge_{p_i \in sel} \neg comp\_total_{p_i}^{k_1,k_2} \right) \Rightarrow \left(\bigvee_{p_j \notin sel} \neg comp\_total_{p_j}^{k_1,k_2} \right) \tag{3.2}$$

The base case in the recursive definition of *MaxKeyDiff* constraints is *MaxKeyDiff* (k_1, k_2, $|P|-1$), defined as shown in (3.3).

$$\bigvee_{p \in P} \neg comp\_total_{p}^{k_1,k_2} \tag{3.3}$$

Given a *MinKeyDiff*(k_1, k_2, n), this scheme leads to $\sum_{i=0}^{n} \binom{|P|}{i}$ clauses of size $|P|$. Given a *MaxKeyDiff*(k_1, k_2, n), this scheme leads to $\sum_{i=|P|-n}^{n} \binom{|P|}{i}$ clauses of size $|P|$ as well.

This scheme was expected to perform well in cases where a customer would prefer a stricter *KeyDiff* constraints, but would accept a solution for less strict one, e.g. a customer wants *MinKeyDiff* between two keys to be 2, but is willing to accept a solution with *MinKeyDiff* 1. There are two ways to implement this, either to compile the problem with *MinKeyDiff* 2 and if the SAT solver finds no solution, compile the problem with *MinKeyDiff* 1 and run the solver again, or use incremental solving.

Incremental solving is the process of adding new clauses and variables to a problem already once solved by a SAT solver. The previous clauses, variables, assignments and inferences are kept between runs, meaning that most of the problem is already solved, in our case this means that the recursive portion of the *KeyDiff* constraint is already present and solved.

3.2.2 "Direct" definition scheme

The alternative scheme is based on principle that if at least k positions out of n should differ, then no group of $N-k+1$ positions can contain only positions that do not differ. This idea can be straightforwardly converted into CNF using *comp_total* variables, i.e. a $MinKeyDiff(k_1, k_2, n)$ is converted as shown in (3.4).

$$\bigwedge_{sel \in selectK(P,|P|-n+1)} \left(\bigvee_{p \in sel} comp\_total_p^{k_1,k_2} \right) \tag{3.4}$$

The same concept applies to conversion of *MaxKeyDiff*. If k positions out of n should not differ, then no group of $N-K+1$ positions can contain only positions where the two keys differ. This means that a $MaxKeyDiff(k_1, k_2, n)$ is converted to CNF as shown in (3.5).

$$\bigwedge_{sel \in selectK(P,|P|-n+1)} \left(\bigvee_{p \in sel} \neg comp\_total_p^{k_1,k_2} \right) \tag{3.5}$$

Given $KeyDiff(k_1, k_2, n)$, this scheme leads to $\binom{|P|}{|P|-n+1}$ clauses of length $|P|-n+1$. This means that at worst, this scheme will generate clauses as long as the generalized legacy scheme, but will generate shorter clauses for stricter constraints. In fact, for *KeyDiff* constraining only a single position, the generated clause will be the same.

This scheme has a very strong advantage in some pathological cases, because a *MinKeyDiff* constraint that allows two keys to only have the same cutting at a single position will only generate $\frac{|P| \cdot |P-1|}{2}$ binary clauses, where the generalized version of the original scheme would generate $\sum_{i=0}^{|P|-1} \binom{|P|}{i}$ clauses with length $|P|$. This saves a significant number of clauses and leads to overall shorter clause length.

Similarly, a highly constraining *MaxKeyDiff* constraint, e.g. a *MaxKeyDiff* that allows two keys to only have different cutting position at a single position, will only generate $\frac{|P| \cdot |P-1|}{2}$ binary clauses.

3.3 Reducing the number of decision variables

As mentioned in section 1.4.2, decision variables enable users to give a SAT solver extra information to guide its selection of variables to branch on.

For our problem, only *key* and *lock* variables describe a real, physical, property of the master-key system, and other variables were created to simplify defining various constraints and logical properties of the system. This means that only *key* and *lock* variables need to be decision variables, and other variables can be be set as non-decision.

Setting *blocking* and *shape* variable as non-decision variables at the same time as defining them using only implications (shown in section 2.4.2) is unsound and an incorrect solution may be found. Therefore there are 5 sound combined configurations, as shown in table 3.1.

decision variables	*blocking* definition	*shape* definition	sound
all	$\Rightarrow$	$\Rightarrow$	$\surd$
all	$\Rightarrow$	$\Leftrightarrow$	$\surd$
all	$\Leftrightarrow$	$\Rightarrow$	$\surd$
all	$\Leftrightarrow$	$\Leftrightarrow$	$\surd$
key, *lock*	$\Rightarrow$	$\Rightarrow$	$\times$
key, *lock*	$\Rightarrow$	$\Leftrightarrow$	$\times$
key, *lock*	$\Leftrightarrow$	$\Rightarrow$	$\times$
key, *lock*	$\Leftrightarrow$	$\Leftrightarrow$	$\surd$

Table 3.1. Overview of possible configurations when combining decision variables with implication/equivalence usage in definitions

3.4 Reformulating profile positions

Keyway profiles are currently treated as special positions, where keys have *KeyDepthLockDepth* mappings and where locks can have at most 1 cutting depth, as seen in (2.8). This approach is fully compatible with the optimization described in section 2.4.1. This section proposes a different way of converting keyway profiles into CNF clauses.

3.4.1 New profile formulation

The new formulation of profile positions is based on the observation that a key can open a lock only if they have compatible profiles and, inversely, that key is blocked in a lock if they have incompatible profiles. More formally, let us define two sets, T and F, $T, F \subseteq K \times L$, where T consists of all (key-, lock-) -profile pairs that open each other, and F consists of all (key-, lock-) -profile pairs that are blocked. As an example, taking the profile map from figure 1.7, we get T and F shown in (3.6).

$$\begin{aligned} T &= (0,0),(0,1),(0,2),(0,3),(1,1),(1,3),(2,2),(2,3),(3,3) \\ F &= (1,0),(1,2),(2,0),(2,1),(3,1),(3,2),(3,3) \end{aligned} \tag{3.6}$$

These definitions mean that $|T| + |F| = d^2$, where d is the number of profiles at a given position.

A key k opening a lock l at a profile position p is compiled into SAT as shown in (3.7).

$$\bigwedge_{(kp,lp)\in F} \neg key^{k}_{p,kp} \vee \neg lock^{l}_{p,lp} \tag{3.7}$$

This means that $|F|$ binary clauses are created for each key, lock pair that open each other, as opposed to d clauses of varying lengths. We can expect $|F| > d$ to hold, especially in larger profile maps, where $|F|$ tends to approach d^2. However, this reformulation can still be advantageous as SAT solvers handle binary clauses much faster than longer ones. More importantly, this formulation is not compatible with the optimization in 2.4.1, as it does not force any cutting depths to be selected in the lock for the profile position.

A seemingly similar observation "a key can open lock only if they have compatible profile", formally expressed as shown in (3.8), was neither used, nor tested, as it was

deemed likely to worsen performance compared to the old formulation. Converting this formulation into CNF would either result in up to $2^{|T|}$ clauses of size $|T|$, using a naive approach and distributing the inner conjunctions, or in $|T|$ new variables and $2 \cdot |T|$ new ternary clauses. This was deemed likely to worsen the performance, because the old formulation of *opens* for profile positions creates d new clauses ($d < |T|$) and no new variables.

$$\bigvee_{(kp,lp) \in T} key^{k}_{p,kp} \wedge lock^{l}_{p,lp} \tag{3.8}$$

We can use similar observation, "a key is blocked in a lock if they do not have compatible profiles", to reformulate how the *block* variable is defined at profile positions

$$block^{k,l}_{p} \iff \bigwedge_{(kp,lp) \in T} \neg key^{k}_{p,kp} \vee \neg lock^{l}_{p,kl} \tag{3.9}$$

Notice that we no longer define a *block* variable for each depth at position, but rather a single variable per profile position[1]. This saves a number of variables equal to the number of profiles per each blocked key-lock pair. As noted in section 2.3.5, the number of blocked key-lock pairs approaches $|K| \cdot |L|$ for large master-key systems, making this potentially significant[2].

To properly analyze the resulting CNF clause sizes, we need to decompose the equivalency into 2 implications, shown in (3.10) and (3.11).

$$block^{k,l}_{p} \Rightarrow \bigwedge_{(kp,lp) \in T} \neg key^{k}_{p,kp} \vee \neg lock^{l}_{p,kl} \tag{3.10}$$

$$block^{k,l}_{p} \Leftarrow \bigwedge_{(kp,lp) \in T} \neg key^{k}_{p,kp} \vee \neg lock^{l}_{p,kl} \tag{3.11}$$

Converting the implication in (3.10) to CNF leads to $|T|$ clauses of size 3, as seen in (3.12). This can be an improvement, because even though the original formulation leads to simpler, binary, clauses, it creates $d + |F|$ of them, and for large profile maps it holds that $|T| \ll |F|$.

$$\bigwedge_{(kp,lp) \in T} \neg block^{k,l}_{p} \vee \neg key^{k}_{p,kp} \vee \neg lock^{l}_{p,lp} \tag{3.12}$$

Converting the implication in (3.11) to CNF leads to $2^{|T|}$ clauses with size $|T| + 1$, generated as a disjunction of Cartesian product over literals from all binary clauses on the right-hand side, disjuncted with the $block^{k,l}_{p}$ variable. The conversion could be made smaller with use of stand-in variables, but doing so would create more variables than were saved by defining only one *block* variable per profile position[3].

Because of this, this formulation is advantageous when *block* variables are defined implicatively, but leads to extremely large number of non-trivial clauses otherwise. In

[1] The final blocking clause remains disjunction over all blocking variables for given key, lock pair.

[2] In fact, even for inputs with small number of profiles the number of created literals was cut roughly in half.

[3] Essentially, it would lead back to the old formulation of profile positions.

order to use it when *block* variables are defined using equivalences, we have to make use of a different observation: "A key is blocked in a lock if they have incompatible profiles". Expressed as a logical expression, it leads to the two implications in (3.13) and (3.14).

$$block_p^{k,l} \Rightarrow \bigvee_{(kp,lp)\in F} key_{p,kp}^k \wedge lock_{p,kl}^l \tag{3.13}$$

$$block_p^{k,l} \Leftarrow \bigvee_{(kp,lp)\in F} key_{p,kp}^k \wedge lock_{p,kl}^l \tag{3.14}$$

Converting the implication in (3.14) to CNF leads to $|F|$ clauses with size 3, as shown in (3.15).

$$\bigwedge_{(kp,lp)\in F} block_p^{k,l} \vee \neg key_{p,kp}^l \vee \neg lock_{p,lp}^l \tag{3.15}$$

However, converting the implication in (3.13) to CNF leads once again to $2^{|F|}$ clauses of size $|F|+1$, for the same reasons as converting the implication in (3.11). What we can do is to take the fact that all equivalences in (3.16) hold and use that to define the *block* variables as shown in (3.17).

$$\begin{aligned} block_p^{k,l} &\iff \bigvee_{(kp,lp)\in F} key_{p,kp}^k \wedge lock_{p,kl}^l \\ block_p^{k,l} &\iff \bigwedge_{(kp,lp)\in T} \neg key_{p,kp}^k \vee \neg lock_{p,kl}^l \\ \bigvee_{(kp,lp)\in F} key_{p,kp}^k \wedge lock_{p,kl}^l &\iff \bigwedge_{(kp,lp)\in T} \neg key_{p,kp}^k \vee \neg lock_{p,kl}^l \end{aligned} \tag{3.16}$$

$$\bigwedge_{(kp,lp)\in T} \neg key_{p,kp}^k \vee \neg lock_{p,kl}^l \Rightarrow block_p^{k,l} \Rightarrow \bigvee_{(kp,lp)\in F} key_{p,kp}^k \wedge lock_{p,kl}^l \tag{3.17}$$

This formulation of *blocking* for profile positions keeps the advantage of defining a single *block* variable per profile position, instead of defining a variable per profile. Clause-wise, it generates $|T|+|F|$ (or d^2) clauses of length 3 when *block* variables are defined using equivalence, and thus can be competitive with the old way of converting profile positions to CNF.

The above can be easily generalized to platforms with multiple profile positions: define clauses and variables for every profile position separately. The reformulations of *opens* and *blocks* are also independent of each other and thus it is possible to e.g. define *opens* using the new formulation, but define *blocking* using the old formulation.

In case of multiple profile positions it is also possible to use different formulation for each profile position, but this has not been implemented and tested in this work.

3.5 Summary

Before summarizing the effect of possible optimizations in master-key system conversion to CNF, we have to bring back terms from the summary of the current state of the compiler in section 2.3.5 and define some new ones.

- p is the number of positions (the size of *depth tuple*)
- nd is the number of all possible cutting depths (or $\Sigma_{i=0}^{p}(d_i + 1)$)
- no is the number of all (key, lock) pairs that open
- nb is the number of all (key, lock) pairs that block
- np is the number of profiles in the platform
- kp is the number of all different key pairs (or $\frac{|K| \cdot |K-1|}{2}$)
- T is the set of all compatible profiles
- F is the set of all incompatible profiles

Optimization	Compatible with	
	Decision variables	Lock clause removal
Reformulated profile *opens*	√	×
Reformulated profile *blocked*	√	√
Implicative *block*	×	√
Implicative *shape*	×	√

Table 3.2. Compatibility overview between different settings

Optimization	CNF size difference	
	adds	removes
Reformulated profile *opens*	$no \cdot \|F\|$ binary clauses	$no \cdot np$ clauses of varying size
Reformulated profile *blocked*	$nb \cdot (\|F\| + \|T\|)$ clauses of size 3 nb variables	$nb \cdot np$ clauses of varying size $nb \cdot np$ variables
Implicative *block*	—	$2 \cdot nb \cdot nd$ clauses with size 2
Implicative *block*: new profiles	—	$nb \cdot \|T\|$ clauses of size 3
Implicative *shape*	—	$\|K\| \cdot d_{max} \cdot p$ binary clauses
Lock clause removal	—	$\|L\| \cdot p$ clauses with sizes d_i
Setting decision variables	N/A	N/A
KeyDiff n: incremental scheme	$kp \cdot \sum_{i=0}^{n} \binom{p}{i}$ clauses of size p	N/A
KeyDiff n: "direct" scheme	$kp \cdot \binom{p}{p-n+1}$ clauses of size $p - n + 1$	N/A

Table 3.3. Summary of the different settings' influence on the final CNF size

Judging only by table 3.3, it would seem that defining stand-in variables as an implication is clearly advantageous. However, it does not necessarily hold that more clauses make a problem harder, especially when the clauses are small. Implicative definitions are also incompatible with exploiting *decision* variables. Similarly, removing the lock clauses specifying that each position in a lock has to have a cutting is not a clear win, because it is incompatible with the reformulated profile *open*.

None of the other optimizations can be clearly decided, either. Since $|F| + |T| = p^2$ we can safely assume that $|F| > p$ and $|T| > p$. This means that reformulating profile positions increases the total number of generated clauses, but also generally creates

shorter clauses. In the case of *blocked* profiles, it also decreases the total number of created variables.

Since none of the approaches is obviously better than others, we compare the possible optimizations using a benchmark detailed in chapter 5.

Chapter 4
Optimizing compiler internals

This chapter goes over the more practically oriented optimizations, such as using different SAT solvers, optimizing memory consumption in both compiler and the solver, investigating possible improvements in the implementation of MinSat and using domain-specific knowledge to provide the solver with hints as to what assignments are more likely for specific variables.

4.1 Hinting assignments of variables

One of the possible avenues towards speeding up solving master-key systems is providing hints as to the likely truth value of specific variables. These hints can have two forms, unary clauses, in which case the SAT solver does not consider other possibilities, and default truth assignment, in which case the solver always considers the hinted assignment first, but can find solutions that require different assignment from the one hinted.

These hints are usually derived from some external domain-specific knowledge, such as the fact that, as mentioned in section 1.2.1, some types of lock-charts are known to be solvable in polynomial time. Another potential source of information is solving a master-key system with reduced lock-chart, and then applying this solution to the same problem with full-sized lock-chart.

The part of the master-key system solver responsible for providing hints shall be referred to as the *hinter*. The compiler can query the hinter for a likely key shape for any key, and then apply the shape to the generated SAT formula (for details see section 4.1.1).

The hinter itself distinguishes between two types of keys, individual keys and master keys[1] and uses different ways of generating shapes for both. A pseudo-code for this is shown in figure 4.1. The profile positions are explicitly excluded from being used by the rotating constant method because their semantics in locks differs from standard positions and their effect on the algorithm is not well-studied.

4.1.1 Applying key shape hint

A *key shape hint* is an n-tuple of numbers specifying cutting depths of the key, excluding profile positions. Assuming that all variables have default assignment of false, a key shape hint for key k is applied by setting specific *key*, *shape* and *lock* variables to true, as shown in (4.1).

$$\begin{aligned} \forall (p,d) \in hint_k &: key^k_{p,d} \\ \forall (p,d) \in hint_k &: shape^k_d \\ \forall (p,d) \in hint_k, \forall l \in openedBy(k) &: lock^l_{p,d} \end{aligned} \tag{4.1}$$

[1] For the purposes of hinter, every key that opens at most 2 locks is considered an individual key.

```
# Step 0: Initialization
lockchart       -- the original lockchart being solved
individual-keys -- set of keys that open at most 2 locks
num_positions   -- number of non-profile positions in the platform
gecons          -- set of {\em gecons}
excons          -- set of {\em excons}

# Step 1: Generating shapes for master keys
lockchart := remove-individual-keys(lockchart)
lockchart := remove-duplicated-locks(lockchart)
pos_used := num_positions
while solve(lockchart, timeout):
    key-hints := extract-solution(lockchart)
    force-keys-same-at(pos_used)
    pos_used := pos_used - 1

# Step 2: Generating shapes for individual keys
for shape in rotating-constant(general-key, num_positions - pos_used):
    if satisfies(shape, gecons, excons):
        key-hints += shape
```

Figure 4.1. Pseudocode of how hinter derives hints for different types of keys

The defaults for *comp* and *comp_total* variables are unchanged, because these variables depend on shapes of two keys, rather than just on the shape of one key, requiring a quadratic amount of work. The defaults for *blocking* variables are unchanged for a similar reason, as their interactions with cuttings for keys and locks is even more complex.

4.2 Memory consumption and SAT variable storage

High memory usage is one of the limiting factors on the size of the master-key problems that can be solved. The high-water mark of memory consumption over all inputs described in chapter 5 is ~70 GB, ~6 GB of which was used by the compiler.

Because a SAT solver views specific variables as numbers, e.g. as "132", while the compiler views them by their purpose, e.g. as $key^{k}_{p,d}$, the compiler needs a way to transform it's own view of variables to that of a SAT solver. This mapping is what takes most of the memory used by the compiler, as it works by allocating a large flat array for each variable type, e.g. a *key* is a type and *lock* is different type, and storing the mapped values there. The index into these arrays is calculated using a common flattening method, e.g. the index for variable $key^{k_1}_{p_1,d_1}$ is calculated as $k_1 \cdot p \cdot d + p_1 \cdot d + d_1$, where p is the number of positions in the platform and d is the largest depth for this platform.

Using the largest depth introduces large inefficiencies for platforms with significantly uneven depth tuples, such as the example shown in section 2.2, where the last position has more than 7× the possible depths of other positions. This results in significant memory-

	p_0	p_1	p_2	p_3	p_4	p_5
d_0	21	23	26	30	33	36
d_1	22	24	27	31	34	37
d_2	•	25	28	32	35	38
d_3	•	•	29	•	•	39
d_4	•	•	•	•	•	40

Figure 4.2. Mapping of single key to SAT variables. Numbers are actual SAT values, dots mark slots wasted due to using maximal depth.

use overhead for these platforms[1]. figure 4.2 provides an illustration of how the memory is wasted and how the mapping is realized for platform with depths (1,2,3,2,2,4).

This section describes 3 possible approaches towards reducing this memory overhead and explains why they were ultimately rejected. The three described approaches are:

- computing the values of SAT solver variables directly, just as the original prototype used to,
- using jagged arrays instead of a single flat one,
- using a data structure intended for mapping sparse keys to values, e.g. a hash map.

The original prototype used a simple scheme that is very similar to the current one, but instead of calculating an index inside an array where the value of corresponding SAT variable is stored, it used this index directly as the variable's number. To disambiguate between different types of compiler variables, an arbitrary ordering was imposed upon the variable types and the computed index of a variable is offset by the number of variables allocated by all preceding variable types.

This scheme has some advantages, such as no need to allocate additional memory inside the compiler, and no need to look up the mapping as it can be computed directly. It does however suffer from the same disadvantage as the current mapping scheme: overhead for platforms with highly different depths at different positions. This disadvantage is further exacerbated by this scheme because the overhead is in the number of variables allocated by the SAT solver and a variable inside SAT solver takes at least 60 bytes of memory, while a slot to store an allocated variable outside of SAT solver only takes 4 bytes. Adding extra variables to a SAT solver also incurs some performance overhead, even if they are not used.

While the overhead could be eliminated by calculating exact offsets per each position and variable, effectively trading CPU time for memory usage,[2] there is another problem that makes the original scheme infeasible. The current version of the compiler supports solving a problem with only part of the lock-chart added, e.g. the compiler might at first only add keys 1 and 5 and try to solve this smaller problem, before adding other keys. A small example of how the current mapping scheme deals with this is shown in figure 4.3.

Using directly-indexed jagged arrays to implement the current approach of storing actual values for specific variables is an easy way to eliminate the overhead from un-even

[1] For one real-world platform this overhead means that only roughly 1 out of 3.3 bytes, allocated for these mappings, is actually used.

[2] This can also be applied towards reducing the memory overhead of the current mapping approach.

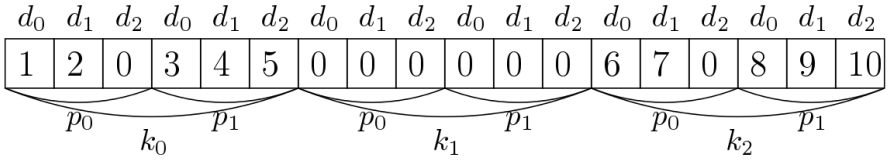


Figure 4.3. Mapping of multiple keys to SAT variables. Numbers are actual SAT values, zeroes mark slots that do not have a value yet.

depths, but has several disadvantages. It increases the number of allocations sharply, e.g. for $comp_{p,d}^{k_1,k_2}$ variable mapping, naively using jagged arrays would require $|K|^2 \cdot |P|$ allocations. Given that each allocation incurs a memory usage overhead, this approach is unlikely to actually decrease memory usage of the variable mapping, and the decreased data locality of this scheme would incur a performance overhead.

The number of allocations could be decreased by using a less naive approach towards using jagged arrays, where only the position dimension is split-off from the flattened mapping and other dimensions remain flattened. This would lead to only $|P|$ allocations per mapping, but would require extra programmer's work for each mapping and would still incur a CPU time overhead because of increased indirection and decreased data locality.[1]

Sparse maps of various kinds run into similar problems, where they decrease data locality and add overhead, but the source of memory overhead is different. Maps need to store the key as well as the value, and the keys in this case are larger than the values, e.g. for $comp_{p,d}^{k_1,k_2}$ the key size is at least 10 bytes[2], most likely 12 bytes due to alignment requirements. This means that to break even, at most 1 out of 4 mapping slots can be used, and that for "even" depth tuples, the memory usage would be quadrupled.

Given that the most un-even platform we have uses at least 1 out of ~3.3 mapping slots, in practice this approach only increases the total memory needed by the compiler to map its own variables to SAT variables.

4.3 Different SAT solvers

The master-key system solver consists of 2 main parts: a compiler, that compiles a master-key system in a set of logical clauses in CNF, and an off-the-shelf SAT solver, that then solves the resulting SAT format. Initially, MiniSat version 2.2 was used, with some extra fixes to address building the used C++ library parts on various platforms. MiniSat was the original choice because it provides an easy-to-use set of bindings for C++ code.

This led us to test Glucose 3.0 as the underlying SAT solver, because it is based on MiniSat 2.2 and its C++ library API is identical to MiniSat's. Glucose improves over MiniSat by introducing of new technique of selecting which learnt clauses are worth keeping, naming the kept clauses "glue clauses".

We also decided to test a solver not based on MiniSat to determine the difficulty with which we could integrate a completely new SAT solver. To this end we decided to test

[1] Iterating over each depth of each position is a common pattern inside the compiler, making this quite costly.

[2] $2 \cdot 4$ bytes for *keys*, 1 byte for position, 1 byte for depth

CryptoMiniSat, because it is under active development and also regularly places highly in the yearly SAT solving competitions. To make run-time switching between different SAT solvers possible, we also had to create an abstraction layer over different SAT solver interfaces.

These three solvers and versions were chosen for testing:

- A fork of MiniSat version 2.2, hash `2f9caab52053ca2498d83ef201e31cbe229da073`[1]
- A fork of Glucose version 3.0, hash `bd3ed96d47c575346519b109b9ad99ca930c4d45`[2]
- CryptoMiniSat version 5.0.1, official release[3]

The MiniSat and Glucose forks do not contain any changes to the core solver functionality, only changes that enable/simplify building them on different platforms.

4.3.1 Reasons for the chosen versions

MiniSat was grandfathered in from the original prototype for the compiler. The exact reasons why it was chosen are unknown, but some speculations can be done: it is well known for being simple, yet performant, SAT solver. This makes it appropriate for prototyping new applications that rely on a SAT solver to do the heavy lifting.

The reason why Glucose was chosen in version 3.0 and not later ones is relatively simple. According to Glucose version 4.0 and 4.1 release notes, the newer versions brought a non-deterministic[12] parallel SAT solver to the table, along with some internal refactoring. This means that the single threaded performance of version 3.0 should be equivalent[4] with the single threaded performance of all the newer versions.

The parallel version of Glucose is not relevant to our use case for two reasons, firstly: it has a non-standard licence, which makes it unclear whether we can use it in commercial product the way we want to, and secondly: we desire determinism, which the parallel version does not offer.

For CryptoMiniSat we picked the last officially released version at the time of implementation[5]. At the time, this version was over a year old and there had been ~1000 commits to the CryptoMiniSat's repository since. This suggests that some potentially significant performance improvements have been done since the release we used, but ~3 commits per day made picking a stable commit to work with hard.

CryptoMiniSat also provides a parallel version, but it does not guarantee deterministic results when used in parallel either[13].

4.3.2 Unified SAT solver API

The unified SAT solver API inside the compiler needs to abstract over the core functionality of SAT solvers and over optional settings. The core functionality includes translating between variable encodings of the compiler and the solver, literal encoding, creating variables and adding clauses to the problem. The optional settings include, but are not limited to, setting the preferred truth value for a variable, the random seed, or the time budget a solver is allowed to use.

[1] `https://github.com/cernoch/minisat`
[2] `https://github.com/horenmar/glucose`
[3] `https://github.com/msoos/cryptominisat/releases`
[4] Modulo changes to how the code ends up being laid-out in the binary by the C++ compiler
[5] At the time of writing, there is a pre-release of version 5.0.2

This abstraction layer is implemented by having a per-solver translator, that provides a translation between the compiler function calls and used data structures to calls to the specific SAT solver and its own data structures. These translators also implement some basic performance optimizations, such as caching memory used to translate compiler's input to the solver's data structures.

4.4 Changing MiniSat's implementation of `lbool`

This change is taken from Mate Soos's blog post[14] about modifying MiniSat's implementation of `lbool`, a class implementing tri-state booleans. He notes that the implementation of `lbool` has changed significantly between MiniSat version 2.0 and 2.2, to remove conditional branches from its operators.

A shortened version of the old implementation is shown in figure 4.4, while figure 4.5 shows a shortened version of the new implementation. Notice that the old implementation has very simple comparison operators, but XORing uses a branch. Because of the context this branch is mostly unpredictable, making it expensive. The new implementation has much simpler implementation of XOR, only a single XOR on the internal value, but has a complex, although branch-free, equality operator.

```
class lbool {
    char     value;

public:
    explicit lbool(int v) : value(v) { }
    lbool()       : value(0) { }
    lbool(bool x) : value((int)x*2-1) { }

    bool  operator == (lbool b) const { return value == b.value; }
    bool  operator != (lbool b) const { return value != b.value; }
    lbool operator ^ (bool b) const {
        return b ? lbool(-value) : lbool(value);
    }
};

const lbool l_True  = lbool( 1);
const lbool l_False = lbool(-1);
const lbool l_Undef = lbool( 0);
```

Figure 4.4. Reduced `lbool` from MiniSat 2.0

Soos's proposed implementation follows the code from MiniSat 2.0, but implements the XOR operator differently, as `return lbool(value * (-2*(char)b + 1));`. This also avoids the branch, while keeping the extremely simple (and thus cheap in CPU time) comparison operators.

Table 4.1 shows optimized ASM output for the three `lbool` implementations, as compiled by GCC 4.9.4 using `-O3` flag. `operator!=` has been omitted for brevity as it is always equal to the equality operator with `sete` changed into `setne` and vice-versa. As it shows, MiniSat 2.2's comparison operators contain more than 10 instructions with

```
class lbool {
    uint8_t value;

public:
    explicit lbool(uint8_t v) : value(v) { }

    lbool(): value(0) { }
    explicit lbool(bool x) : value(!x) { }

    bool  operator == (lbool b) const {
        return !!(((b.value&2) & (value&2)) |
                 (!(b.value&2)&(value == b.value)));
    }
    bool  operator != (lbool b) const { return !(*this == b); }
    lbool operator ^  (bool  b) const {
        return lbool((uint8_t)(value^(uint8_t)b));
    }
};

#define l_True  (lbool((uint8_t)0))
#define l_False (lbool((uint8_t)1))
#define l_Undef (lbool((uint8_t)2))
```

Figure 4.5. Reduced `lbool` from MiniSat 2.2

a moderately long dependency chain, but the XOR operator is trivial. In contrast, both MiniSat 2.0 and the proposed implementation have trivial comparison operators, but longer XOR operator. In case of MiniSat 2.0, it contains only 3 instructions, but one of them is an unpredictable branch. The newly proposed implementation requires 5 instructions, but they are all part of a dependency chain.

Thanks to the lack of unpredictable branching, the newly proposed implementation can be reasonably expected to be faster than the old one from MiniSat 2.0. However, because of the significantly different trade-offs with the implementation in MiniSat 2.2, whether the proposed implementation performs better needs to be measured.

	operator==	operator^

MiniSat 2.0	`cmp BYTE PTR [rdi], sil`
	`sete al`
	`ret`
		`test sil, sil`
		`jne .L7`
		`movzx eax, BYTE PTR [rdi]`
		`ret`

MiniSat 2.2	`movzx eax, BYTE PTR [rdi]`
	`mov edx, esi`
	`xor ecx, ecx`
	`shr dl`
	`xor edx, 1`
	`cmp al, sil`
	`sete cl`
	`and eax, esi`
	`and edx, ecx`
	`and eax, 2`
	`or eax, edx`
	`setne al`
	`ret`
		`mov eax, esi`
		`xor al, BYTE PTR [rdi]`
		`ret`

Proposed	`cmp BYTE PTR [rdi], sil`
	`sete al`
	`ret`
		`movsx edx, BYTE PTR [rdi]`
		`movzx esi, sil`
		`neg esi`
		`lea eax, [rsi+1+rsi]`
		`imul eax, edx`
		`ret`

Table 4.1. An overview of optimized assembly output for different `lbool` implementations. Compiled with GCC 4.9.4, using −O3

Chapter 5
Results

5.1 Benchmarking setup

All benchmarks were run using a dedicated departmental server. The server had two CPU sockets, each with a Intel Xeon E5-2687W 0 clocked at 3.10GHz. As main memory, there was 128 GB of DDR-3 RAM clocked at 1600 MHz. The operating system was Debian 9.2 (Stretch), with Linux kernel in version 4.9.0-3-amd64, compiled using gcc 6.3.0.

Each input in each configuration was given 3600 seconds before being killed by an external watchdog. All randomness inside the SAT solvers was disabled in order to get repeatable results.

5.2 Settings and configurations

As the result of this work, 7 settings for the compiler were implemented and benchmarked. An overview of these settings is shown in table 5.1. One extra setting that was implemented but was not benchmarked is the two alternate ways of translating *KeyDiff* constraints. This is due to all of the inputs being treated identically by the two implementations; none of the inputs use a *MaxKeyDiff* constraint and the *MinKeyDiff* constraints, whenever used, specify a difference of 1, where both implementations give the same set of clauses.

Setting	Possible values
shape variable	Use implication/equivalence in definition
SAT solver	MiniSat, Glucose, CryptoMiniSat
hinter	Use / Do not use
blocking variable	Use implication/equivalence in definition
Decision variables	All / only *key* and *lock* variables
opens for profile positions	Old / New
blocked for profile positions	Old / New

Table 5.1. Overview of settings and their possible values.

Doing a simple cartesian product over all settings would suggest that there are 192 possible configurations, but some options are incompatible with each other, namely implicative definition of either *shape* variables or *blocking* variables cannot be used when the set of decision variables does not encompass all variables. Consequently, there are only 120 valid configurations for the compiler.

5.2.1 Configuration names

Each configuration can be described by a septuple containing values assigned to all of the compiler settings. The order of the settings is the same as used in table 5.1. Throughout this chapter, specific configurations will be referred to by a name consisting of comma-separated internal names for each individual setting in the configuration.

The internal, more compact, name for each setting is shown in table 5.2.

Value	internal name
Define *shape* variable using implication	impli-shape
Define *shape* variable using equivalence	equiv-shape
Use MiniSat as the underlying SAT solver	minisat
Use Glucose as the underlying SAT solver	glucose
Use CryptoMiniSat as the underlying SAT solver	cmsat
Use hinter	hinter
Do not use hinter	no-hinter
Define *blocking* variable using implication	impli-block
Define *blocking* variable using equivalence	equiv-block
Set all variables as decision variables	all-decisions
Set only *key* and *lock* variables as decision variables	reduced-decisions
Use new formulation for profile position *opens*	new-profile-opens
Use old formulation for profile position *opens*	old-profile-opens
Use new formulation for profile position *blocked*	new-profile-blocks
Use old formulation for profile position *blocked*	old-profile-blocks

Table 5.2. Overview of internal names for different setting values.

As an example, a configuration where

- *shape* variables are defined using equivalence,
- CryptoMiniSat is used as the SAT solver,
- hinter is used,
- *blocking* variables are defined using equivalence,
- reduced set of decision variables is used,
- the new formulation of opening at profile positions is used,
- the old formulation of blocking at profile positions is used,

would be named `equiv-shape/cmsat/hinter/equiv-block/reduced-decisions/new-profile-opens/old-profile-blocks` using the names from table 5.2.

5.3 Inputs

All inputs used in master-key system solver benchmarking were provided by our industry partners. Consequently, the measured results should reflect the performance of our compiler when used in practice. On the other hand, this also means that the exact specifics of platform descriptions contained within are confidential and thus cannot be described in this work. Only approximations that attempt to duplicate the interesting properties of the inputs can be provided here. Their names cannot be used either, so

they will be referred to as “Manufacturer A” through “Manufacturer F”. In total, 46 inputs were provided from 6 different key manufacturers, but after filtering out trivial inputs[1], only 34 inputs were left.

Per-Manufacturer Breakdown is shown in table 5.3.

Manufacturer	# inputs	# inputs used
Manufacturer A	8	7
Manufacturer B	11	9
Manufacturer C	3	3
Manufacturer D	6	1
Manufacturer E	15	12
Manufacturer F	3	2

Table 5.3. Overview of manufacturers and number of inputs provided by them

Different manufacturers also use different platforms, sometimes significantly so. An overview of the basic properties is shown in table 5.4. With the exception of Manufacturer A, all inputs provided by a single manufacturer use the same platform. Because Manufacturer A tends to have a high number of profiles, the average depth per position is provided in two numbers. The first is the average over standard positions, the one in parentheses is average including profile positions.

Manufacturer	# positions	avg. depth at position	# profiles
Manufacturer A (3 inputs)	6	4	0
Manufacturer A (1 inputs)	6	4 (8)	1
Manufacturer A (3 inputs)	6	4 (12)	1
Manufacturer B	12	6	0
Manufacturer C	17	6	12
Manufacturer D	7	7	1
Manufacturer E	6	9	0
Manufacturer F	30	3	0

Table 5.4. Overview of manufacturers’ platform properties

The number of *excons* and *gecons* also varies significantly between different manufacturers, with the least constrained platform having only ~60 constraints and the most constrained platform having ~300 constraints. A per-platform breakdown is shown in table 5.5.

More detailed descriptions of each manufacturer’s platform follows. Note that, because of the aforementioned confidentiality agreement, the descriptions have to be kept somewhat vague and inexact.

5.3.1 Platform description

Manufacturer A provided inputs with 3 different platforms, two of which use profile positions and one does not. For platforms that contain profile positions, the profiles

[1] An input was considered trivial if it contained less than ~50 keys or locks.

Manufacturer	# *excons*	# *gecons*
Manufacturer A	150	10
Manufacturer B	60	0
Manufacturer C	180	120
Manufacturer D	180	0
Manufacturer E	90	120
Manufacturer F	70	0

Table 5.5. Approximate number of *excons* and *gecons* per manufacturer

are assigned manually by the manufacturer. This means that the input does not exercise the compiler ability to select profiles All of the Manufacturer A's platform uses *KeyDepthLockDepth blocked* mappings for non-profile positions.

Manufacturer B's platform does not use profile positions and has the least number of constraints. The largest provided input, at ~2000 keys and locks, uses this platform.

Manufacturer C's platform has the most profile positions of all platforms, with 12 profile positions. Discounting profile positions, this platform has the least cutting positions, and is also the most constrained platform. Manufacturer C is also one of the two manufacturers that use *KeyDepthLockDepth* mappings for non-profile positions.

Only one of Manufacturer D's inputs was classified as non-trivial and used for benchmarking. The input uses 1 profile position, but each key has a manually assigned profile from the manufacturer.

Manufacturer E's platform is on-par with Manufacturer A's platform as the shortest, but has the highest number of possible cutting depths per position and does not utilize keyway profiles. It is the second most constrained platform.

Manufacturer F's platform has a very large number of positions, but each position has only few possible cutting depths. There are no profile positions and the overall number of constraints is low.

5.4 Evaluation methodology

Every input from section 5.3 has been run against every configuration from section 5.2 and the total run time of the compiler was recorded. The measured run time for an input is then compared to the run time of the *baseline* configuration, and the relative result is then used for further comparisons. Two configurations are compared by taking an average of these relative results over relevant inputs. As an example, if baseline configurations took 20, 100 and 900 seconds to solve problems A, B and C respectively, and configuration A needed 30, 80 and 800 seconds to solve these problems, then configuration A's relative times would be 1.5, 0.8 and 0.89 and the average would be 1.06 meaning that configuration A is worse than the baseline.

The baseline configuration is a set of settings that most closely resemble the state of the compiler before this work started, described as `impli-shape` / `minisat` / `no-hinter` / `impli-block` / `no-reduced-decision-vars` / `old-profile-opens` / `old-profile-blocks` using rules from section 5.2.1.

If the compiler exceeds 3600s when solving an input, a run time of 4000s is used.

Configuration	Avg. of rel. run times
impli-shape/minisat/no-hinter/impli-block/all-decisions/old-profile-opens/new-profile-blocks	0.973
impli-shape/minisat/no-hinter/impli-block/all-decisions/new-profile-opens/old-profile-blocks	0.974
equiv-block/minisat/no-hinter/impli-block/all-decisions/new-profile-opens/old-profile-blocks	0.975

Table 5.6. Best 3 configurations when compared over inputs from all manufacturers

5.5 Benchmark results

The results of the three best configurations over all inputs are shown in 5.6. As shown, the best configurations provide a small but measurable improvement over the baseline.

Comparing configurations for each manufacturer's platform separately[1] shows that tuning configuration towards specific platform provides significant speedup for 3 manufacturers, Manufacturer A, Manufacturer C and Manufacturer D, as shown in tables 5.7, 5.8 and 5.9 respectively.

Configuration	Avg. of rel. run times
equiv-shape/minisat/no-hinter/equiv-block/reduced-decisions/old-profile-opens/old-profile-blocks	0.729
equiv-shape/minisat/no-hinter/impli-block/all-decisions/old-profile-opens/new-profile-blocks	0.953
impli-shape/minisat/no-hinter/impli-block/all-decisions/old-profile-opens/new-profile-blocks	0.954

Table 5.7. Best 3 configurations when compared over all inputs from Manufacturer A.

Manufacturer A's platform shows a significant speed-up, 27%, but for a single specific configuration only. Interestingly, the 3 worst results all have the average run time 460× longer than the baseline. This shows that there is a significant potential for difference in performance when using different configurations, but the baseline happens to already be near the optimum for this platform.

Interestingly, all 3 worst-performing configurations for this platform use Glucose with blocking variables defined using equivalence and with hinter disabled.

The best configurations for Manufacturer B's platform improve upon the baseline's run time by less than 2%, while the worst-performing configurations increase the run time ~40×. This suggests that there is less space for performance differences between different configurations and that the baseline configuration is near the optimum again.

Once again, all 3 worst-performing configurations for this platform use Glucose with blocking variables defined using equivalence and with hinter disabled.

Manufacturer C's platform shows a significant benefit from different configurations, with the best configuration improving over the baseline by ~50 %. The worst-performing

[1] Manufacturer A's 3 platforms are considered together, as they are very similar with the exception of keyway profiles.

Configuration	Avg. of rel. run times
equiv-shape/minisat/hinter/equiv-block/reduced-decisions/new-profile-opens/old-profile-blocks	0.507
equiv-shape/glucose/no-hinter/equiv-block/reduced-decisions/old-profile-opens/old-profile-blocks	0.552
impli-shape/glucose/no-hinter/impli-block/all-decisions/new-profile-opens/new-profile-blocks	0.561

Table 5.8. Best 3 configurations when compared over all inputs from Manufacturer C

Configuration	Avg. of rel. run times
equiv-shape/glucose/no-hinter/equiv-block/reduced-decisions/new-profile-opens/new-profile-blocks	0.335
equiv-shape/glucose/no-hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	0.362
impli-shape/glucose/no-hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	0.363

Table 5.9. Best 3 configurations when compared over all inputs from Manufacturer D

configurations increase the run time ~16× and, interestingly, all use MiniSat as the SAT solver, together with equivalence-defined block variables.

Results for Manufacturer D's platform need to be taken with a grain of salt, as Manufacturer D provided us with only a single non-trivial input to benchmark. However, because the underlying platform remains the same even for trivial inputs, the results can be expected to generalize well across other non-trivial inputs using the same platform.

Manufacturer E's and Manufacturer F's platforms have very similar results. The best-performing configurations do not improve performance significantly and the distance between best and worst-performing configuration is small for both (from 0.958 to 6.069 and from 0.973 to 7.317 respectively).

These two platforms also share common parts of the worst-performing configurations; they both use hinter and use CryptoMiniSat as the underlying SAT solver.

5.5.1 Closer look at the effect of individual settings

This section is primarily based on the results shown in section 5.5, but also discusses results that were omitted from the main work for brevity and can be found in appendix B.

When looking at global and per-platform results, some trends in regards to individual settings emerge. One of them is that configurations using the hinter are rarely amongst the best-performing, but they also are not overly represented amongst the worst-performing configurations. This is caused by the fact that the run time cost of hinter's step 1 (as described in figure 4.1) is significant and increases with the cost of obtaining a full solution of the whole problem.

Using the hinter has proven to be a win in one regard: only 8 out of all configurations have been able to find solution to the largest input tested (described in section 5.3.1),

with all of them utilizing the hinter. Because the input does not use profiles, the 8 configurations collapse into 2:

- `impli-shape/minisat/hinter/equiv-block/reduced-decisions`,
- `equiv-shape/minisat/hinter/equiv-block/reduced-decisions`.

Among SAT solvers, MiniSat is the best-performing solver by far. It is used by all 3 best configurations globally and the 3 best configurations for five of the platforms. It is also used by the best-performing configuration for the sixth one. The remaining SAT solvers, Glucose and CryptoMiniSat, are both widely used by the worst per-platform configurations and Glucose is used by all 3 of the globally worst configurations. Unlike CryptoMiniSat, Glucose does have a platform it excels at solving, specifically Manufacturer C's platform.

Glucose's underperformance relative to MiniSat is likely caused by the fact that it uses significantly different search restarts that are optimized towards solving unsatisfiable problems, rather than the satisfiable ones[15].

The different ways of defining *shape* variables do not seem to have a clear effect on the performance of the compiler. This is an interesting result on its own, as the difference between using implicative definition and equivalence-based definition is adding a large amount of binary clauses. This ought to have a measurable effect on the results and further investigations need to be done.

Reducing the number of decision variables enables a significant speed-up for Manufacturer A's and Manufacturer C's platforms, but also enables the worst-performing configurations for Manufacturer B's platform.

There are no discernible trends for the performance impact of different ways of defining *block* variables. While the implicative definition is used by the globally best configurations and equivalence-based definition by the globally worst, the per-platform results are more mixed, i.e. both definitions show up about as often in the best configurations as in the worst-performing configurations.

Out of the 6 manufacturers, only 3 use keyway profiles in their platforms and out of these three, two assign profiles to keys manually. Consequently, there is a very limited sample size for evaluating the impact of different conversions of profile positions to SAT.

To judge performance for platforms without manually-assigned profiles we should look at the results for Manufacturer C's platform. There, the best configuration uses the new definition for opening and the old one for blocking at profile positions. At the same time, the second best uses the old definitions for both and the third one uses new definitions for both, and all three configurations improve upon the baseline significantly. What is also interesting is that there is no option shared between all three of these successful configurations. This suggests that both schemes for defining opening and blocking at profile positions can perform equally well when used for platform where keys are not manually assigned to their profiles, but highly depend on other configurations.

There is one combination that seems to perform badly; the three worst-performing configurations for Manufacturer C's platform share the combination of using new definitions for blocking but old definitions for opening at profile positions.

5.6 Evaluating changes to MiniSat implementation

The results discussed above do not include the modified version of MiniSat because the changs to MiniSat's implementation of `lbool` should not significantly change MiniSat's speed. After-all, the main parts, e.g. the branching heuristic, remain the same, only the speed of a small part of internal functions changes. Because MiniSat has already shown the best performance amongst the tested SAT solvers, the potential speed-up would not change the results, reported in 5.5, in a meaningful way.

To evaluate the performance of the modifications, we compared the modified MiniSat to unmodified MiniSat 2.2 across different inputs and configurations. The configurations used are the same as described in section 5.2, but the inputs were only a subset of those used for comparing different configurations for the compiler. The run times of both versions for each configuration, input pair were then compared, and cases when either of the two MiniSat versions performed better by more than a certain threshold were counted.

The results for three different thresholds are shown in table 5.10. As they show, for most problems the difference between the two is insignificant, but sometimes one version is statistically significantly faster. However, the unmodified implementation wins by more than a specific threshold more often, meaning that its run time is better on average.

Threshold	# MiniSat-mod better	# MiniSat-2.2 better	Runs compared
2%	31	292	932
5%	17	42	932
10%	11	21	932

Table 5.10. Summary of MiniSat modification benchmark results

Chapter 6
Conclusion

This thesis investigated performance characteristics of a SAT-based master-key system solver developed by the Department of Computer Science, FEE, CTU. It described the current state of the solver, along with explaining what factors change the size of the generated SAT problem and detailing what optimizations for this process already existed before this work started.

Further possible changes to the conversion to SAT, along with more practically-oriented optimizations, were proposed and benchmarked. The practically-oriented optimizations included employing domain-specific knowledge to guide the underlying SAT solver towards finding the solution, and changing the implementation of MiniSat's internals.

Benchmarking these proposed changes showed that no set of changes is significantly beneficial over all inputs provided by our industrial partners, but that most platforms can be sped-up significantly by using the right configuration for the compiler. Two configurations also enabled the compiler to solve a very large, and previously unsolved, master-key system in ~10 minutes.

Based on the results, there are several possible directions for future work. First is to obtain permission and submit SAT problems generated by the compiler to SAT solving competitions. Classical wisdom suggests that MiniSat is significantly outperformed by newer SAT solvers, but for our specific SAT problems it performed the best out of the three tested solvers. This suggests that the SAT output from the compiler has some properties that are not well exploited by the more modern SAT solvers.

Another is to explore new heuristics for the hinter component of the compiler. The current implementation of the hinter allows the solver to solve previously unsolvable problems and limits the performance penalty from some of the worst configurations, but its own run-time cost pessimises total run time of simple inputs. There are several places where a new heuristic could provide a significant speed-up. First, a heuristic to determine at which position the step 1 should start could avoid several costly calls to the SAT solver. Second, a heuristic could be used to provide solutions for general keys without invoking the SAT solver and third, a better heuristic for setting the timeout of the SAT solver in step 1 could also provide significant time savings.

Third direction for future work is to select which profile definition is used by using a cost model, rather than the current model where the definition is decided by a command-line flag. The variables that indicate the number and size of clauses generated by converting profile positions to SAT, e.g. $|F|$ and $|T|$, are either known or easy to calculate, so dynamically selecting conversion scheme should carry only small run-time penalty. There is even potential for significant gains for platforms with multiple profile positions where a different, and hopefully better performing, conversion scheme could be chosen on a per position basis. However, it is important to note that because the cause-and-effect relationship between generated problem size and a SAT solver's performance is non-trivial, any such decision making would be only heuristical.

Finally, there is one non-speculative improvement that can be implemented in the compiler. As was explained in chapter 2, the two dominant factors influencing the size of resulting SAT problem are implementation of *blocking* relations and *shape* variables used by *KeyDiff* constraints. However, for keys with different *opens* sets on platforms without profile positions, the problem structure already guarantees that a *MinKeyDiff* of 1 between these two keys is satisfied. Because this is the most common kind of a *KeyDiff* constraint, skipping converting *MinKeyDiff* constraints under these conditions should prove a worthwhile optimization.

In closing, solving complex master-key systems is still a very hard and mostly unexplored problem that deserves further attention, if not for anything else, then for the real-world application it has. Regrettably, we cannot release our industrial input portfolio nor can be our implementation open-sourced, but the description contained within this thesis should provide any interested party a basis towards implementing their own master-key system solver and experimenting with it, further exploring this field.

References

[1] Černoch, Radomír. *Lock-chart solving*. Czech Technical University in Prague, 2017. Ph.D. Thesis. Unpublished, can be found at https://github.com/cernoch/mks-dis.

[2] James, Peter, and Nick Thorpe. *Ancient Inventions*. New York: Ballantine Books, 1994. ISBN 978-0345364760.

[3] *Mul-T-Lock CLIQ sales page*. http://www.mul-t-lock-cliq.com/.

[4] O'Shall, Don. *The Definitive Guide to RCM – Rotating Constant Method of Master Keying*. Locksmithing Education, 2015. ISBN 9781937067137. https://books.google.cz/books?id=5Hz_rQEACAAJ.

[5] Cook, Stephen A. The Complexity of Theorem-proving Procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 1971. pp. 151–158. STOC '71. Available from DOI 10.1145/800157.805047. http://doi.acm.org/10.1145/800157.805047.

[6] Davis, Martin, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. *Commun. ACM*. New York, NY, USA: ACM, jul, 1962, Vol. 5, No. 7, pp. 394–397. ISSN 0001-0782. Available from DOI 10.1145/368273.368557.

[7] Eén, Niklas, and Niklas Sörensson. An extensible SAT-solver. In: *Theory and applications of satisfiability testing*. 2003. pp. 502–518.

[8] Audemard, Gilles, and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In: *Proceedings of the 21st International Jont Conference on Artifical Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. pp. 399–404. IJCAI'09.

[9] Tseitin, G. S. On the Complexity of Derivation in Propositional Calculus. In: Jörg H. Siekmann, and Graham Wrightson, eds. *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983. pp. 466–483. ISBN 978-3-642-81955-1. Available from DOI 10.1007/978-3-642-81955-1_28. https://doi.org/10.1007/978-3-642-81955-1_28.

[10] Selman, Bart, Henry A Kautz, Bram Cohen, and others. Local search strategies for satisfiability testing.. *Cliques, coloring, and satisfiability*. 1993, Vol. 26, pp. 521–532.

[11] Selman, Bart, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. In: *Proceedings of the Tenth National Conference on Artificial Intelligence*. AAAI Press, 1992. pp. 440–446. AAAI'92. ISBN 0-262-51063-4.

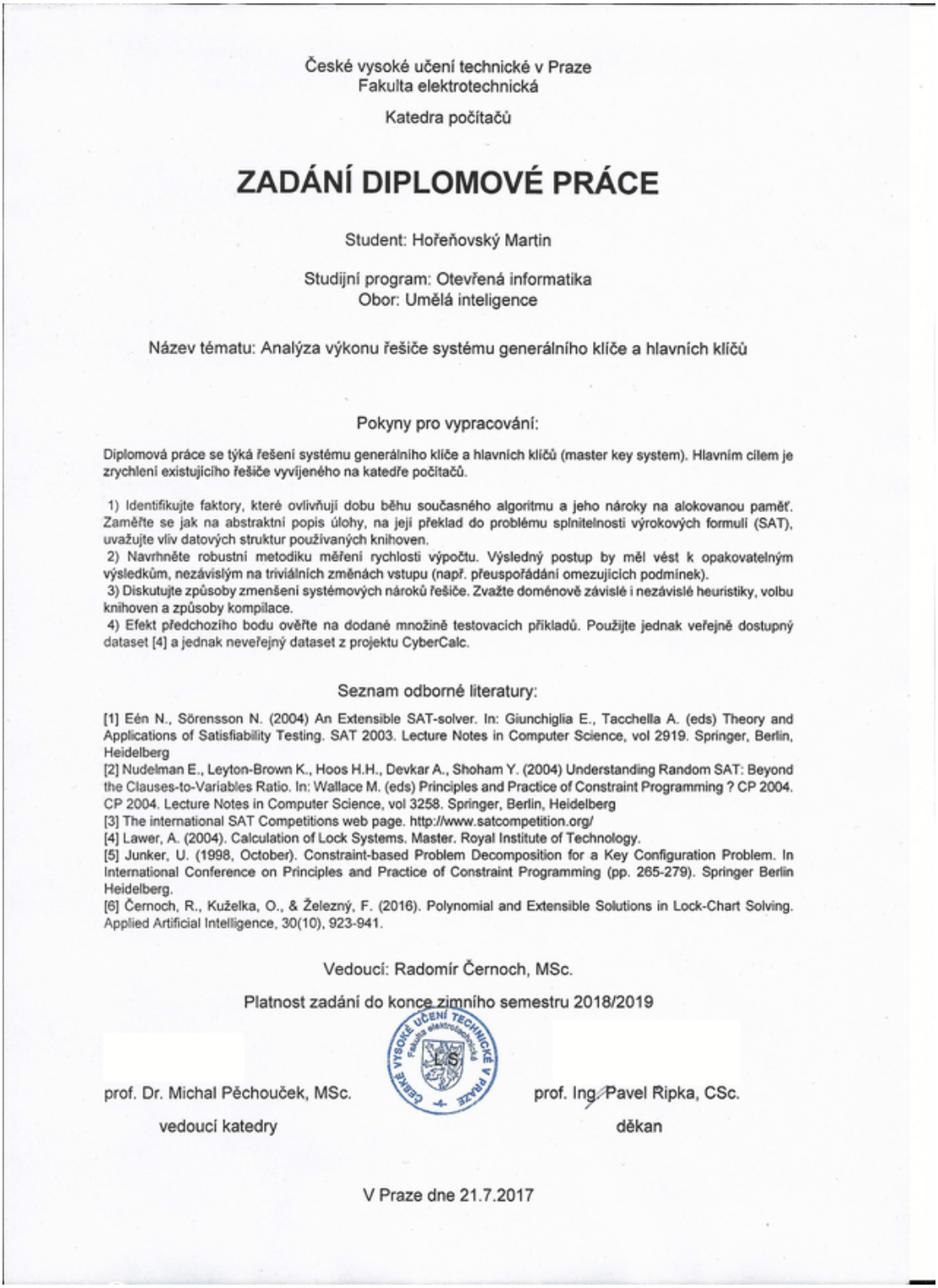
[12] *Release notes for Glucose 4.0*.
http://www.labri.fr/perso/lsimon/glucose/.

[13] *Mate Soos's answer to "Are cryptominisat's results deterministic when using multiple threads?" issue on github*.
https://github.com/msoos/cryptominisat/issues/443#issuecomment-354576602.

[14] *Mate Soos's blog post about lbool in MiniSat*.
https://www.msoos.org/2014/03/speeding-up-minisat-with-a-one-liner/.

[15] Oh, Chanseok. *Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL*. New York University, 1, 2016. Ph.D. Thesis.

Appendix A
Specification



České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Hořeňovský Martin

Studijní program: Otevřená informatika
Obor: Umělá inteligence

Název tématu: Analýza výkonu řešiče systému generálního klíče a hlavních klíčů

Pokyny pro vypracování:

Diplomová práce se týká řešení systému generálního klíče a hlavních klíčů (master key system). Hlavním cílem je zrychlení existujícího řešiče vyvíjeného na katedře počítačů.

1) Identifikujte faktory, které ovlivňují dobu běhu současného algoritmu a jeho nároky na alokovanou paměť. Zaměřte se jak na abstraktní popis úlohy, na její překlad do problému splnitelnosti výrokových formulí (SAT), uvažujte vliv datových struktur používaných knihoven.
2) Navrhněte robustní metodiku měření rychlosti výpočtu. Výsledný postup by měl vést k opakovatelným výsledkům, nezávislým na triviálních změnách vstupu (např. přeuspořádání omezujících podmínek).
3) Diskutujte způsoby zmenšení systémových nároků řešiče. Zvažte doménově závislé i nezávislé heuristiky, volbu knihoven a způsoby kompilace.
4) Efekt předchozího bodu ověřte na dodané množině testovacích příkladů. Použijte jednak veřejně dostupný dataset [4] a jednak neveřejný dataset z projektu CyberCalc.

Seznam odborné literatury:

[1] Eén N., Sörensson N. (2004) An Extensible SAT-solver. In: Giunchiglia E., Tacchella A. (eds) Theory and Applications of Satisfiability Testing. SAT 2003. Lecture Notes in Computer Science, vol 2919. Springer, Berlin, Heidelberg
[2] Nudelman E., Leyton-Brown K., Hoos H.H., Devkar A., Shoham Y. (2004) Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In: Wallace M. (eds) Principles and Practice of Constraint Programming ? CP 2004. CP 2004. Lecture Notes in Computer Science, vol 3258. Springer, Berlin, Heidelberg
[3] The international SAT Competitions web page. http://www.satcompetition.org/
[4] Lawer, A. (2004). Calculation of Lock Systems. Master. Royal Institute of Technology.
[5] Junker, U. (1998, October). Constraint-based Problem Decomposition for a Key Configuration Problem. In International Conference on Principles and Practice of Constraint Programming (pp. 265-279). Springer Berlin Heidelberg.
[6] Černoch, R., Kuželka, O., & Železný, F. (2016). Polynomial and Extensible Solutions in Lock-Chart Solving. Applied Artificial Intelligence, 30(10), 923-941.

Vedoucí: Radomír Černoch, MSc.

Platnost zadání do konce zimního semestru 2018/2019

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 21.7.2017

Appendix B
Full results

Chapter 5 discusses the best and worst-performing configurations and displays the best-performing ones for platforms with significant speed-ups. This appendix provides tables of the best and worst-performing configurations for all platforms.

For practical reasons, this appendix cannot contain all of the measured data. Instead an anonymized .csv file is provided on the enclosed CD, and online, at `https://codingnest.com/files/thesis-results.csv`

Configuration	Avg. of rel. run times
`impli-shape/minisat/no-hinter/impli-block/all-decisions/old-profile-opens/new-profile-blocks`	0.973
`impli-shape/minisat/no-hinter/impli-block/all-decisions/new-profile-opens/old-profile-blocks`	0.974
`equiv-shape/minisat/no-hinter/impli-block/all-decisions/new-profile-opens/old-profile-blocks`	0.975
`impli-shape/glucose/no-hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks`	97.536
`equiv-shape/glucose/hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks`	97.651
`impli-shape/glucose/hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks`	97.654

Table B.1. 3 best and worst configurations when compared over all inputs

Configuration	Avg. of rel. run times
`equiv-shape/minisat/no-hinter/equiv-block/reduced-decisions/old-profile-opens/old-profile-blocks`	0.729
`equiv-shape/minisat/no-hinter/impli-block/all-decisions/old-profile-opens/new-profile-blocks`	0.953
`impli-shape/minisat/no-hinter/impli-block/all-decisions/old-profile-opens/new-profile-blocks`	0.955
`impli-shape/glucose/no-hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks`	461.845
`equiv-shape/glucose/no-hinter/equiv-block/all-decisions/new-profile-opens/old-profile-blocks`	462.116
`impli-shape/glucose/no-hinter/equiv-block/all-decisions/new-profile-opens/old-profile-blocks`	462.143

Table B.2. 3 best and worst configurations when compared over all inputs from Manufacturer A

Configuration	Avg. of rel. run times
equiv-shape/minisat/no-hinter/equiv-block/all-decisions/new-profile-opens/old-profile-blocks	0.981
equiv-shape/minisat/no-hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	0.982
equiv-shape/minisat/no-hinter/equiv-block/all-decisions/new-profile-opens/new-profile-blocks	0.984
equiv-shape/glucose/no-hinter/equiv-block/reduced-decisions/old-profile-opens/new-profile-blocks	40.561
equiv-shape/glucose/no-hinter/equiv-block/reduced-decisions/new-profile-opens/new-profile-blocks	40.576
equiv-shape/glucose/no-hinter/equiv-block/reduced-decisions/new-profile-opens/old-profile-blocks	40.657

Table B.3. 3 best and worst configurations when compared over all inputs from Manufacturer B

Configuration	Avg. of rel. run times
equiv-shape/minisat/hinter/equiv-block/reduced-decisions/new-profile-opens/old-profile-blocks	0.508
equiv-shape/glucose/no-hinter/equiv-block/reduced-decisions/old-profile-opens/old-profile-blocks	0.552
impli-shape/glucose/no-hinter/impli-block/all-decisions/new-profile-opens/new-profile-blocks	0.562
equiv-shape/minisat/no-hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	16.182
impli-shape/minisat/hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	16.203
equiv-shape/minisat/hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	16.282

Table B.4. 3 best and worst configurations when compared over all inputs from Manufacturer C

Configuration	Avg. of rel. run times
equiv-shape/glucose/no-hinter/equiv-block/reduced-decisions/new-profile-opens/new-profile-blocks	0.335
equiv-shape/glucose/no-hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	0.362
impli-shape/glucose/no-hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	0.363
impli-shape/minisat/hinter/impli-block/all-decisions/old-profile-opens/new-profile-blocks	1.457
equiv-shape/cmsat/hinter/equiv-block/reduced-decisions/new-profile-opens/old-profile-blocks	1.461
equiv-shape/cmsat/hinter/equiv-block/reduced-decisions/new-profile-opens/new-profile-blocks	1.506

Table B.5. 3 best and worst configurations when compared over all inputs from Manufacturer D

Configuration	Avg. of rel. run times
equiv-shape/minisat/no-hinter/equiv-block/all-decisions/new-profile-opens/new-profile-blocks	0.958
equiv-shape/minisat/no-hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	0.969
equiv-shape/minisat/no-hinter/impli-block/all-decisions/old-profile-opens/old-profile-blocks	0.973
equiv-shape/cmsat/hinter/impli-block/all-decisions/old-profile-opens/new-profile-blocks	6.013
impli-shape/cmsat/hinter/equiv-block/all-decisions/new-profile-opens/new-profile-blocks	6.041
impli-shape/cmsat/hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	6.070

Table B.6. 3 best and worst configurations when compared over all inputs from Manufacturer E

Configuration	Avg. of rel. run times
impli-shape/minisat/no-hinter/impli-block/all-decisions/new-profile-opens/old-profile-blocks	0.973
impli-shape/minisat/no-hinter/equiv-block/all-decisions/old-profile-opens/new-profile-blocks	0.992
impli-shape/minisat/no-hinter/equiv-block/all-decisions/new-profile-opens/old-profile-blocks	0.993
equiv-shape/cmsat/hinter/impli-block/all-decisions/old-profile-opens/new-profile-blocks	7.258
equiv-shape/cmsat/hinter/equiv-block/all-decisions/new-profile-opens/new-profile-blocks	7.264
impli-shape/cmsat/hinter/impli-block/all-decisions/old-profile-opens/old-profile-blocks	7.317

Table B.7. 3 best and worst configurations when compared over all inputs from Manufacturer F

Appendix C
Glossary

binary clause	A clause consisting of 2 literals
CDCL	Conflict Driven Clause Learning is a modification of the DPLL algorithm with non-chronological backtracking
CNF	Conjuctive normal form. A logical formula is in CNF if it is a conjunction of clauses, which themselves are disjunctions of literals
DNF	Disjunctive normal form. A logical formula is in DNF if it is a disjunction of clauses, which themselves are a conjuction of literals
DPLL	David-Putnam-Logemann-Loveland algorithm is a complete backtracking search algorithm for solving (CNF) SAT
equi-satisfiable	Two formulae are equi-satisfiable if either both formulae can be satisfied, or neither can
pin tumbler lock	A tumbler lock where spring-loaded pins are used to block the tumbler from moving
SAT	Abbreviation of Boolean Satisfiability Problem — determining whether given boolean formula can be satisfied
ternary clause	A clause consisting of 3 literals
tumbler	A part of lock that blocks it from opening until the correct key is inserted
unit clause	A clause with single (unassigned) literal
3-SAT	SAT variant where each formula is limited to at most 3 literals