THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : VERIMAG

Analyses quantitatives pour les attaquants adaptatifs

Quantitative analysis for adaptive attackers

Présentée par :

Thomas VIGOUROUX

Direction de thèse :

Marius BOZGA INGENIEUR DE RECHERCHE HDR, CNRS DELEGATION ALPES	Directeur de thèse
Laurent MOUNIER MAITRE DE CONFERENCE, UNIVERSITE GRENOBLE ALPES	Co-encadrant de thèse
Cristian ENE MAITRE DE CONFERENCES, UNIVERSITE GRENOBLE ALPES	Co-encadrant de thèse

Rapporteurs :

DANIEL LE BERRE
PROFESSEUR DES UNIVERSITES, UNIVERSITE D'ARTOIS
SYLVAIN CONCHON
PROFESSEUR DES UNIVERSITES, UNIVERSITE PARIS-SACLAY

Thèse soutenue publiquement le **2 décembre 2024**, devant le jury composé de :

LYDIE DU BOUSQUET, PROFESSEURE DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Présidente
DANIEL LE BERRE, PROFESSEUR DES UNIVERSITES, UNIVERSITE D'ARTOIS	Rapporteur
SYLVAIN CONCHON, PROFESSEUR DES UNIVERSITES, UNIVERSITE PARIS-SACLAY	Rapporteur
VALERIE VIET TRIEM TONG, PROFESSEURE, CENTRALESUPELEC	Examinatrice
SUBHAJIT ROY, ASSOCIATE PROFESSOR, INDIAN INSTITUTE OF TECHNOLOGY, KANPUR	Examinateur

Invités :

MARIUS BOZGA
INGENIEUR DE RECHERCHE HDR, CNRS DELEGATION ALPES
LAURENT MOUNIER
MAITRE DE CONFERENCES, UNIVERSITE GRENOBLE ALPES

Abstract

Evaluating the security of a program is a notoriously difficult task, but of paramount importance considering the prevalence of computer systems in today's world.

A possible path towards security evaluation proceeds in steps: first search for vulnerabilities or bugs within the program, then evaluate how to turn said bugs into profit, whether it be information or capabilities with respect to the program. In order to perform security evaluation of the program, one has thus to consider what is called an *attacker model*. We propose in this thesis an *attacker hierarchy*, with increased reasoning capabilities as we go up the hierarchy: *non-adaptive*, *adaptive*, or *multiple* adaptive attackers. We study the elements of the hierarchy depending on the attacker's objective, whether it be triggering a bug or gaining information.

While qualitative criteria for evaluating the security of a program exist, their quantitative counterparts are relatively less studied. The interest behind quantitative criteria is twofold: one could easily *compare* two programs in order to, for example, assess that a given vulnerability has been addressed properly, or to schedule bugs to be fixed in priority; or one could provide a quantitative goal that the program has to reach which depends on the target application domain, and enforce the security policy this way.

The goal of this thesis is thus to develop new quantitative criteria for program security, based on quantitative attacker's objectives, which yields so-called *counting problems*. Counting problems are, as their name implies, problems related to computing the size of a set. We focus our interest on one counting problem for each level of the hierarchy: Max#SAT for non-adaptive, SSAT for adaptive, and DQMax#SAT for multiple attackers. In this thesis, we define formally two quantitative attacker's objectives, and develop encodings of said attacker's objective as a counting problem. We then resolve these combinatorial optimization problem, which corresponds to answering the question "how easy is it for the attacker to fulfill this goal?".

Our contributions are the following, at each level of the attacker hierarchy:

1. First, we present a formal framework to reason about programs running against an attacker, as well as encodings of quantitative security criteria within this framework. We also present reductions of said security criteria to counting problems.
2. Then, new resolution methods for the counting problems used within our framework are presented, motivated by resolution in practice. This includes approximation schemes and algorithms in order to circumvent the high practical complexity of these problems.
3. Implementation of our resolution methods, as well as experimental evaluation and comparison to other existing methods on security-related benchmarks as well as benchmarks specific to the counting problem studied.

Remerciements

I would first of all like to thank the jury for their time reviewing this work. I would especially like to thank the reviewers, Daniel Le Berre and Sylvain Conchon, for their detailed review, as well as their remarks and corrections to the report.

En français maintenant, j'aimerais remercier mes encadrants pour le temps qu'ils m'ont accordé pendant ces trois ans, mais aussi les innombrables relectures de ce manuscrit. Tout d'abords Marius pour le temps passé, que cela soit avant ou après sa prise de rôle officielle, dans des discussions qui ont permis de structurer cette thèse et de la pousser jusqu'à son état actuel. Laurent ensuite, pour les nombreuses discussions, qu'elles soient scientifiques ou non, qui ont permis là aussi d'avancer dans cette thèse, mais aussi pour son rôle dans les moments pivots et son soutient. Enfin Cristian, pour m'avoir poussé à considérer d'autres aspects de la sécurité des programmes, mais aussi pour son soutient dans les moments difficiles de la thèse. À vous trois, sachez qu'avoir la chance de faire cette thèse avec vous a été un plaisir et un honneur. Multumesc (va puteti imagina ca nu am invatat romana)!

Ensuite, sur le plan personnel, je voudrais d'abord remercier mes parents, pour avoir m'avoir soutenu durant tout le processus qu'est une thèse, mais aussi dans les étapes plus folkloriques que cette thèse a pu traverser.

Enfin, je voudrais remercier Ludivine. Parce que derrière ces trois années de travail, il y a aussi trois années à me supporter tous les jours, supporter les "sursauts de motivations". Malgré ça, nous avons traversé les étapes, et le futur semble nous réserver encore de magnifiques moments, à plusieurs maintenant.

Contents

Part I

Introduction

Chapter 1

Context, goals, and challenges

1.1 Context

Computer programs are now present in most of the critical aspects of our life: medical records, public transportation and banking just to name a few. This inevitably leads to the question of the security of such systems: who would entrust their life to a potentially faulty computer system ? Fortunately, some industries (e.g. aerospacial and railways) are already accustomed to thorough verification of their computer systems, mostly because any failure in the software of these systems leads to catastrophic outcomes [Tra19]. This kind of computer systems, i.e. those that we rely on for everyday life, are deemed *critical* by most people.

The place of computer programs makes them a good entry-point for *attackers*. The motivations of an attacker, as well as its capacities, varies greatly depending on its general goal. From extorting a few dollars to destabilisation of a country, the means in order to reach the goal will be vastly different. However, in all cases, the attacker will always leverage software failures in order to reach its goal.

The root cause of program failures are called *bugs*. A bug is a quirk in the program's behavior that the programmer does not expect nor know about. Bugs can originate from many sources [Dul20]: programming mistake, misunderstanding of a programming language feature, error in the translation (*miscompilation*), hardware failure. While bugs are unfortunate from the user's point of view, they are very fortunate from an attacker's point of view.

Exploitation of a bug is the process of turning a bug into profit: whether it be information, or gaining capacity to exploit future bugs, yielding more profit. Discovery and exploitation is thus the *objective* of the attacker, as this is the key to reaching its goal. An exploitable bug is often called a *vulnerability*. Evaluating the dangerousness of a vulnerability is a hard to specify problem: if a bug can be triggered in 90% of the cases bug is not easily exploitable, is it critical ? What about 0.1% chance of triggering the bug, but results in leaking all the secrets of the program ?

We can thus see that the evaluation of the security of a program is two-fold: (1) discovering bugs early in the development process (2) evaluating the *exploitability* of said bug. This is where *program analysis* comes into play. Program analysis is a set of techniques (formal or not) which goal is to explore the possible states of a given program. This state space exploration can be used to find bugs and/or determine if they are exploitable. Many techniques geared towards bug finding exist: abstract interpretation [CC77] can prove the absence of bugs, fuzzing [Zhu+22] tries finds bugs quickly, symbolic execution [CDE08; Dav+16] attempts to explore all the paths of the program to search for bugs. Note though that because of Rice's theorem [Ric53], this search is fundamentally flawed: it is not possible to have algorithms for bug-finding and security evaluation that work in all cases.

In the software security context, program analysis thus tries to enforce a *security policy* based on the attacker model. We can differentiate two kinds of security policies: *qualitative* and *quantitative*. Qualitative security policies corresponds to a yes/no question. They tend to be simpler to enforce in practice, e.g. one could say that the presence of a memory error in the program breaks the security policy. One can more generally say that, that whenever a given property does not hold, the policy fails [Dul20]. On the other hand, quantitative security policies give a score to the program. One can then *compare* the security of programs based on this score. For example one can compute the number of bits leaked by the program on a secret [Smi09]. Quantitative security policies allow more fine-grained evaluation of the security, as well as formal criteria to compare two programs.

1.2 Problems

Based on the context set above, we define the following problems of interest in this thesis.

Advanced attackers The various definitions of security consider different *attacker models*. Attacker models are formal objects that represent the capacities of the attacker. A given program analysis method is obviously tied with a given attacker model, which implies that *more powerful* attacker models will imply *stricter* security guarantees. Many orthogonal capacities can be associated to craft a given attacker model:

- **what is its winning condition ?** Maybe the attacker wants to reach a point in the execution of the program ? Maybe it wants to learn about the program's secrets ?
- **how much can it interfere with the program's state ?** Can it inject faults ? If so, what kind of faults (data, test inversion, micro-architectural) ? Can it provide inputs to the program ?
- **can it learn by interacting with the program ?** That is, is it necessary to consider the state of the knowledge of the attacker together with the state of the program during analysis ?

Many examples of such attacker models exist in the literature. Ducousso, Bardin and Potet [DBP23] present a new symbolic execution method based on an attacker model that can inject hardware faults in the program during execution, with a simple *reachability objective*. Girol, Farinier and Bardin [GFB21] define an alternative symbolic execution based analysis that considers an attacker that carefully crafts an input value once at the start of the program with a more advanced *robust reachability* objective. Finally, Smith [Smi09] considers an attacker whose objective is to learn the secrets of the program.

Security evaluation As mentioned earlier, evaluating the security of a given program amounts to finding a bug, and evaluating its exploitability. However, even the *definition* of exploitability is somewhat fuzzy in the literature [Dul20; Gir22; Smi09]. In order to define what a *secure* program is, one has to first define what exploitability means it one's context. Implicitly, this in turns implies assumptions on the attacker model considered. A good way to evaluate the security of a program is to take the attacker model first, and ask: does there exist a winning strategy for an attacker following this model ? This is and instance of the well-known *synthesis* problem.

1.3 Proposal

In this thesis, we propose security criteria based on attacker models coupled with quantitative objectives. Our goal is to show that the synthesis of the optimal attack strategy is related to established problems, leading to synthesis of said strategies in practice. We thus devise three key contributions in order to perform the security evaluation of a given program.

Simple interactive programs We define a simple imperative language, meant to be a formal *intermediate representation* for programs, on which we can define quantitative security evaluation faithfully. This formal language contains explicit handling of interactions between the program and the attacker, allowing to formally track these interactions during program execution. Furthermore, we define semantics of this formal language focused on the *quantitative evaluation* of the security of the program *with regard to an attacker.*

Quantitative attacker objectives We propose a formulation of attacker objectives based on *probabilities* that unifies previously defined notions of *quantitative reachability* [Gir22] and *leakage* [Smi09]. Unifying these definitions allows for treatment of both issues within the same framework. Furthermore, motivated by practical resolution we show that these security criteria reduce to so-called *counting problems* [Tod91; Tor91], a class of problems related to counting the number of elements in a set, and generalisations as combinatorial optimization problems.

The attacker hierarchy Orthogonally we define a hierarchy of attacker models, varying their *reasoning power* as we traverse the hierarchy. We consider 3 levels in this hierarchy:

1. **non-adaptive**: the attacker is not allowed to exploit the program's outputs to adapt its strategy. It is only allowed to provide the appropriate inputs computed *a priori* to the program in order for it to run, and evaluate their efficiency.

2. **adaptive**: the attacker is allowed to interact with the program and learn from these interactions. The inputs that the attacker provides thus depend on the previous outputs from the program, two different executions may lead to different inputs from the attacker.

3. **multiple adaptive**: this level represents multiple attackers that share a common goal, but do not necessarily share information during the execution of the program. In this case, note that the optimal strategy of the group might not be the combination of the individually optimal strategies.

1.4 Contributions and layout

We order the contributions of this thesis following the layout of the different chapters:

- In Part II we introduce our formalism for programs, as well as a new framework that unifies leakage and quantitative reachability objectives. Chapter 5 presents normalization techniques for the introduced language to avoid path explosion problems. Chapter 11 defines an extension of the aforementioned language for multiple attacks.

- Parts III to V each study one of level of the attacker hierarchy (resp. non-adaptive, adaptive and multiple). Each part follows the same organization:

  1. We first introduce the specificities of the optimal attack synthesis problem at this level, as well as the corresponding counting problem (Chapters 6 and 9 and Section 12.2). We then prove the equivalence between the synthesis problem and the counting problem.
  2. Then, we propose resolution methods for the counting problem (Chapters 7 and 13). These resolution methods are driven by practical resolution, and rely on approximation schemes to achieve this goal.
  3. Finally we perform experimental evaluation of the proposed resolution method, on examples extracted from meaningful security problems (Chapters 8, 10 and 14).

Chapter 2

Notations

Part of the contributions of this thesis is to reduce security problems to counting problems over boolean formulas. We thus state in this chapter base definitions useful for this objective. We begin by definitions relative to the classical boolean logic and associated problems. We then introduce some notions about program analysis in the context of security.

2.1 Boolean logic

We use the standard notations for boolean logic: $\mathbb{B} \triangleq \{\top, \bot\}$ denotes the set of boolean values, $\land$ denotes conjunction, $\lor$ denotes disjunction, $\neg$ denotes negation, $\Rightarrow$ denotes implication and $\Leftrightarrow$ denotes equivalence. Given a formula ϕ we write $\phi(V)$ to denote the fact that the variables appearing in ϕ are a subset of V. We denote as $\mathcal{F}\langle V\rangle$ the set of formulas with variables in V. Given a set of boolean variables V, a literal is a variable or its negation and we denote as $\overline{V}$ the set of literals over V, for example $\overline{\{x, y\}} = \{x, \neg x, y, \neg y\}$.

Valuations are defined as functions from variables to boolean values, that is a valuation v is of type $v : V \to \mathbb{B}$. Valuations are lifted to formulas in the usual way, and we say that v satisfies a formula $\phi(V)$ (that we denote $v \models \phi$) whenever $v(\phi) = \top$. Conversely, we define the *coproduct* of a formula $\phi(V)$ by an assignment $v : V' \to \mathbb{B}$ as the replacement in ϕ of all variables $x \in V'$ by $v(x)$.

We define the restriction $v{\downharpoonright_E}$ of an assignment $v : V \to \mathbb{B}$ to a set $E \subseteq V$ in the usual way for function restrictions. v is then called a *complete* assignment while $v{\downharpoonright_E}$ is called a *partial* assignment.

We say that a formula ϕ is *satisfiable* if there exists v such that $v \models \phi$. Otherwise, ϕ is deemed *unsatisfiable*. Determining whether a formula is satisfiable or unsatisfiable is called the *boolean satisfiability* problem, also known as SAT, and is the prototypical NP-complete problem.

Definition 2.1 (Models of a formula)**.** Given propositional formula $\phi(V)$ and $E \subseteq V$, $\mathcal{M}_E(\phi) \triangleq \{v{\downharpoonright_E} \mid v \models \phi\}$ denotes the *set of models of* $\phi(V)$ *projected over* E.

We omit the set E when it is equal to V, that is: $\mathcal{M}(\phi) \triangleq \mathcal{M}_V(\phi)$.

Definition 2.2 (Entailment)**.** Given two formulas $\phi(V)$ and $\psi(V)$, we say that ϕ *entails* ψ (denoted $\phi \models \psi$) whenever:

$$\mathcal{M}(\phi) \subseteq \mathcal{M}(\psi)$$

Definition 2.3 (Equivalence class)**.** Given a valuation $v : V \to \mathbb{B}$ and a set of variables $E \subseteq V$, we call *equivalence class of* v *over* E the set of valuations that agree with v over E, that is:

$$[v]_E \triangleq \{v' \mid v'\downharpoonleft_E = v\downharpoonleft_E\}$$

The elements of $[v]_E$ are called the *extensions* of $v\downharpoonleft_E$.

2.1.1 Normal forms, clauses and cubes

Normal forms are prevalent in the SAT solving community. They can be categorized based on the computational complexity of the operation one can perform on them (checking satisfiability, counting the number of models, …), as different representations of the same boolean function will have different properties. Normal forms of boolean formulas and their properties are the main interests of the field of *knowledge compilation* and we refer the reader to [DM11] for more informations about that subject.

We introduce here two basic normal forms that are of interest in the SAT community.

Definition 2.4 (Conjunctive normal form (CNF))**.** A formula ϕ is in *conjunctive normal form* if it is a conjunction of disjunctions of literals. A disjunction of literals is called a *clause*.

Example 2.1.1. The following is the CNF representation of the formula $a \Leftrightarrow b$:

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

It contains two clauses:

$$a \vee \neg b \qquad\qquad \neg a \vee b$$

Definition 2.5 (Disjunctive normal form (DNF))**.** A formula ϕ is in *disjunctive normal form* if it is a disjunction of conjunctions of literals. A conjunction of literals is called a *cube* or a *monomial*.

Example 2.1.2. The following is the DNF representation of the formula $a \Leftrightarrow b$:

$$(a \wedge b) \vee (\neg a \wedge \neg b)$$

It contains two cubes:

$$a \wedge b \qquad\qquad \neg a \wedge \neg b$$

Remark 2.1.3. These two normal forms are called *flat*, meaning that the formula has only two levels (e.g. conjunction then disjunction for CNF). CNF are commonly used in SAT solving thanks to the so-called *resolution rule* allowing to syntactically derive new clauses from the set of clauses of the formula, and it is possible to prove unsatisfiability whenever an empty clause is derived.

2.2 Information theoretic notions

We provide in this section a few examples of measures on probability distributions that will be useful in our setting. We place ourselves in the context of probability theory, and use the following notations:

- $\mathbb{D}(V)$ is the set of discrete distributions over a finite domain V
- $\mathbb{P}[E]$ is the probability of an event $E \subseteq V$

We begin by a definition of *vulnerability* of a distribution. If we take a given distribution $D \in \mathbb{D}(V)$, and view it as the distribution of the values of a secret, the vulnerability represents the probability that an attacker guesses the secret value in one try, more formally:

Definition 2.6 (Vulnerability)**.** Given a distribution $D \in \mathbb{D}(V)$, the *vulnerability* of V is defined as:

$$\mathbb{V}[D] \triangleq \max_{v \in V} \mathbb{P}[D = v]$$

Another measure for distributions is the well-known Shannon entropy, which measures the *uncertainty* associated with a given probability distribution. It is measured in *bits*.

Definition 2.7 (Shannon entropy)**.** Given a distribution $D \in \mathbb{D}(V)$, the *Shannon entropy* is defined as:

$$\mathbb{H}_S[D] \triangleq \sum_{v \in V} -\mathbb{P}[D = v] \log \mathbb{P}[D = v]$$

Another kind of entropy is so-called *Rényi entropy*, which is used in cryptography to measure the efficiency of guessing attacks.

Definition 2.8 (Rényi entropy)**.** Given a distribution $D \in \mathbb{D}(V)$ and α such that $0 < \alpha < \infty$ and $\alpha \neq 1$, the *Rényi entropy* is defined as:

$$\mathbb{H}_\alpha[D] \triangleq \frac{1}{1-\alpha} \log\left(\sum_{v \in V} \mathbb{P}[D = v]^\alpha\right)$$

The Rényi entropy is a generalisation of the Shannon entropy and it is related to the vulnerability as exposed in the following two properties. Property 2.1 serves as the definition of *min-entropy*, a measure that will be of interest in the context of quantitative information flow analysis and that is tied to *vulnerability* (Definition 2.6). On the other hand Property 2.2 links Rényi entropy to Shannon entropy.

Property 2.1 (Min-entropy)**.** *Given a distribution $D \in \mathbb{D}(V)$ we have:*

$$\lim_{\alpha \to \infty} \mathbb{H}_\alpha[D] = -\log \mathbb{V}[D]$$

Property 2.2. *Given a distribution $D \in \mathbb{D}(V)$ we have:*

$$\lim_{\alpha \to 1} \mathbb{H}_\alpha[D] = \mathbb{H}_S[D]$$

Finally, mutual information captures measures the amount of dependence between two distributions. We begin be defining conditional entropy, which represents the entropy of a given distribution assuming the value of another is known.

Definition 2.9 (Conditional entropy)**.** Given two distributions $D_1 \in \mathbb{D}(V_1)$ and $D_2 \in \mathbb{D}(V_2)$, we define the conditional entropy as:

$$\mathbb{H}_\alpha[D_1|D_2] \triangleq \sum_{v_2 \in V_2} \mathbb{P}[D_2 = v_2]\mathbb{H}_\alpha[D_1|D_2 = v2]$$

We can then define *mutual information* which is a quantification of the information shared between two distributions.

Definition 2.10 (Mutual information)**.** Given two distributions $D_1 \in \mathbb{D}(V_1)$ and $D_2 \in \mathbb{D}(V_2)$, we define their *mutual information* as:

$$\mathbb{I}[D_1, D_2] \triangleq \mathbb{H}_S[D_1] - \mathbb{H}_S[D_1|D_2]$$

Chapter 3

Related work

We discuss in this chapter the work related to ours. The idea is to provide a broad overview of the field, both related to quantitative software analysis, as well as the specific counting problems investigated throughout this thesis.

3.1 Program analysis

Program analysis' primary concern is to verify that programs actually do what they should. This thus implies the pre-existence of two descriptions: (i) the actual behavior of the program (ii) its expected behavior. The *formal language* used to represent the behaviors is important and is the source of a field called *formal methods* [CDE08; Dav+16; Dul20; OHe20].

For the sake of conciseness, and as this is not the main point of interest on this thesis, we present only a subset of techniques in this section.

In general, the semantics of programs can be represented as *transition systems*, that is, a combination of a set S of *possible configurations* and a *transition relation* T on that set of configurations. Given this representation of programs, we can determine the behavior of the program as being the set of possible *executions* (sequences of configurations using through the transition relation) it exemplifies.

Now, given the behavior of the program, one can perform various verification tasks, each corresponding to some kind of *expected behavior* of the program. Recall that Rice's theorem [Ric53] theoretically prevents the automatic verification (hence enforcement) of arbitrary properties, and specifically security properties.

3.1.1 Testing

A crude technique to check that the program follows the expected behavior is to perform *manual testing.* This amounts to run the program for a number of times and a set of inputs that covers a set of executions that is considered enough by the developer (or the verification team).

The problem with this technique is the actual lack of coverage of the program's state space. Indeed, tests are often written based on the program's source code, following the understanding the programmer has of the program. Unfortunately, the actual behavior of the program and its *understood* behavior may differ, leading to subtle behavior divergences. This difference in understanding is the source of so-called *weird machines* [Dul20].

This verification method follows the *original semantics* of the program, that is, it attempts to cover the state space of the original transition system as defined above. The next methods we will present change the state space of the program, in an attempt to ease verification.

3.1.2 Abstract interpretation

Abstract interpretation [CC77] is a broad field that encompasses techniques for software verification based on alternative configuration spaces for the program called *abstract domains*. The key idea behind abstract domains is to represent *over-approximations* of the concrete state space of the program, such that said over-approximations are *easier* to work with. The objective is to provide semantics relying on the over-approximating capabilities of the abstract domain, such that state space exploration is easier, and avoid the exponential blowup in the number of states of the program. In some case it is even possible to handle possibly non-terminating programs, which strictly enhances the verification capabilities of the analysis.

Here, the targeted verification tasks are often safety of the program: verifying that a bad state *cannot be reached*. Indeed, because of the over-approximation, if no abstract state intersects with the set of bad states, we have proven that the program itself cannot reach said state.

3.1.3 Symbolic execution

Symbolic execution is an implementation of abstract interpretation. The idea behind symbolic execution [CDE08; Dav+16] is to trade the *concrete* state space for a *symbolic* one. What we mean is that we define the semantics of the program as expressions over symbolic variables (e.g. the inputs of the program), and the state of the program is now a *symbolic store*, storing the symbolic expressions that corresponds to each variable.

This shift in representation of the program's behavior can be done such that any concrete state of the program corresponds to a symbolic state of the program. Conversely, each symbolic state corresponds to a set of concrete states. Thanks to that, assuming that one can also convert the expected behavior into a set of symbolic executions, it is possible to verify that the program follows the expected behavior while limiting the size of the state space.

One advantage of symbolic execution is that it allows computing so-called *path predicates*, which are characterisations (in the forms of formulas) of the states that reach a given point in the program. This allows the decoupling of the analyses: one can use a symbolic execution engine to extract path predicates, and then perform the analysis based on this characterisation of the behaviors of the program.

3.1.4 Quantitative robustness

Building on *symbolic execution* [Dav+16], Girol introduces a quantitative variant of reachability named *quantitative robustness* [BG22]. This model includes the attacker's capabilities in the analysis, by splitting the variables of the program in two classes: *controlled* and *uncontrolled* variables. This variant then measures how easily an attacker can reach a point in a program based on this controllability information, and it relies on counting problems.

We view programs as transition relations, where $T(P)$ denotes the set of all the executions (i.e. sequences of states) the program can exemplify. Quantitative robustness therefore assumes two parameters:

- the set a of *controlled variables* that the attacker controls (and knows), the complement of which the attacker neither has control nor knowledge of;
- the set O of *objective executions* that designates winning executions of the program.

The question is then to compute the values for *controlled* variables such the set of winning executions $T(P) \cap O$ is the biggest possible in terms of cardinality. The *efficiency* of a given controlled *value* is defined as the density of the set of winning executions, relativised to the set of all possible uncontrolled variables values. Quantitative robustness is then defined as the maximum of said efficiency.

Bardin and Girol [BG22] then focus on encoding the computation of the quantitative robustness using MAX#SAT[1], focusing on finite executions and finite domains for programs, and proposes an algorithm for resolution of the problem based on knowledge compilation.

A first limitation of this work is the attacker model. In the proposed setting, the attacker does not interact with the program, and thus does not learn from its interactions. This is a major limitation in terms of attacker capabilities, but allows for easier resolution, and lower complexity.

Second, while it is listed as future works, this work does not account for other quantitative measures based on the winning objective. For example, one could consider measuring the amount of information leaked to the attacker. Though, this would require a more explicit handling of the inputs and outputs within the program formalism.

Section 4.4.2 will provide a definition of this concept for use in this thesis. Our definition will extend the notion of quantitative reachability to other levels of our attacker hierarchy: from only non-adaptive attackers in this case, we will extend to adaptive and multiple adaptive attackers. This effectively generalises the notion by allowing for interactivity.

Specific contribution about MAX#SAT will be discussed in Section 3.3 so that comparison with the state of the art is easier.

3.1.5 Incremental attack synthesis

The paper [Sah+21] is another approach to automatically generate the inputs of a program in order to reach some goal. The goal set here is to maximize the amount of information leaked about a secret.

The idea is to consider an interactive program, which both takes inputs from the attacker and outputs information. The objective of the attacker is to pick inputs each time it is asked to, such that it maximizes the amount of information leaked by the program. The state of the program is here again split between inputs (called *low* variables) and secrets (called *high* variables). The outputs are split from the program state and handled separately.

The analysis proceeds in two steps: 1. statically analyze the program to extract so-called *observation constraints* 2. generate the attack incrementally based in these constraints. Observation constraints corresponds to the constraints that the attacker *learns* on the program's state based on the observation that occurred. These constraints link input (low) values and secret (high) values.

[1]It is said in the paper that the underlying problem is F-E-MAJSAT, while the definition used is that of MAX#SAT. Note that while these are equivalent *in terms of computational complexity*, they ask to compute two different things.

These observation constraints are used to learn more about the secret variables of the program, in an incremental way. This means that at each interaction with the program, the analysis will update its state, adding the observation constraint associated with the latest observation. Using this newly updated state the analysis will generate a new input value, with the goal to maximize the information learned at the next interaction.

The problem of generating an input that maximizes the information leakage is very similar to that of synthesizing the inputs for maximizing quantitative robustness. The paper proposes a technique based on simulated annealing to determine such input. The idea is to first compile the current set of constraints into an automaton, such that the size of the language of the automaton is exactly the number of models of the formula (this is call automata-based model counting [ABB15]). This *faster* model counting technique is used in a simulated annealing setting in order to quickly evaluate if the current candidate is better or worst than the current best candidate. As an optimisation, it is possible to incrementally refine the model-counting automaton to at each interaction, avoiding costly reconstructions of the complete automaton.

The primary limitation of this approach is that it is not *symbolic*. What we mean is that one cannot reconstruct the *strategy* that one has to employ in order to maximize the information leakage. The only way to do that is to run the synthesis for each possible value for the secrets, and record all the resulting interactions. This in turn builds an *attack tree* which corresponds to the strategy. Note that this also implies that synthesizing such a strategy can be computationally hard: rerunning the analysis completely for each possible secret value is exponential in the number of bits of the variables.

Second, the technique used for synthesis of the answer at a given interaction (e.g. simulated annealing) has no *formal guarantee* of correctness. This means that one has actually no guarantee about the optimality of the synthesized strategy. It is possible to use techniques such as Max#SAT (Section 3.3) in order to improve the quality of the answer, as well a provide formal correctness and optimality guarantees.

3.1.6 Quantitative information flow

This body of work [Smi09] is interested in quantifying the amount of information leaked by a given program. Especially, it is interested in ways to apply *security policies* based on the quantification of leakage.

In this setting, program are viewed as information theoretic *channels*: they take as input a distribution on secret variables and output a distribution on *low* variables known to the attacker. The goal of the field is to provide measures to assess the security of the channel (i.e. ensuring that it does not leak too much), and this is done using information theoretic measures such as the Shannon entropy.

Extension of this framework were defined to account for interactivity with the programs, but they (at the time of writing) are purely theoretical formulation, without practical implementation [Mes19].

We will define in Section 4.4.3 a version of quantitative information flow that also accounts for interactivity. Also, throughout Part IV, we will show practical resolution methods for this problem.

3.2 Model counting and projected model counting

We mentioned the SAT problem before, which is the most well-known example of NP-complete problems. A strict generalisation of this is the #SAT problem, asking the find the number of models of a formula.

Definition 3.1 (Model counting)**.** Given a propositional formula $\phi(V)$, the *model counting* (#SAT) query denoted by

$$\text{Я}V.\ \phi$$

asks to compute $|\mathcal{M}(\phi)|$.

This problem is obviously strictly harder than the SAT problem, as it is enough to check that the number of models is at least 1.

It is possible to extend the definition of #SAT to another problem called projected model counting. Informally, we then ask to count the number of *partial* models (i.e. restrictions of models) instead of *complete* models.

Definition 3.2 (Projected model counting)**.** Given a propositional formula $\phi(Y, Z)$, the *projected model counting* (#∃SAT) query denoted by

$$\text{Я}Y.\ \exists Z.\ \phi$$

asks to compute $|\mathcal{M}_Y(\phi)|$.

Remark 3.2.1 (Complexity of #SAT and #∃SAT). The class (in terms of computational complexity) of problems asking to compute the number of solutions of a decision problem in a class $\mathcal{C}$ is denoted[a] $\#\cdot\mathcal{C}$. More formally, given $R(x) \in \mathcal{C}$, we have:

$$\#\cdot R \triangleq |\{x \mid R(x)\}|$$

Following this fact, it is easy to see that #SAT is $\#\cdot\mathsf{P}$-complete [HV95; Val79]. Given a boolean formula ϕ take $R(x) \triangleq$ "$x \models \phi$". $R(x)$ is obviously in P, and thus counting the number of models or ϕ is in $\#\cdot\mathsf{P}$, and is actually complete.

Furthermore, #∃SAT is $\#\cdot\mathsf{NP}$-complete [DHK00]. This is because it asks to count the number of assignments *that can be extended* into satisfying assignments, e.g counting the number of y such that $\exists z.\ \phi(y, z)$, which is NP-complete.

Some open questions remain about the relationship of these two classes. Given a #∃SAT query $\text{Я}Y.\ \exists Z.\ \phi$, if $Z = \emptyset$ then the query amounts to model counting; however if $Y = \emptyset$, the problem amounts to checking satisfiability. While we obviously have $\#\cdot\mathsf{P} \subseteq \#\cdot\mathsf{NP}$, it is believed (but unproven) that this relation is strict.

[a] We use the definitions of Toda [Tod91] here, instead of that of Valiant [Val79]. This is because these definitions more closely distinguish the problems like #SAT and #∃SAT compared to Valiant's [HV95].

We continue going up in the generalisation of SAT. We have presented two *quantitative* generalisations of SAT (namely #SAT and #∃SAT), and we will now present more general problems allowing combinatorial optimization on top of these.

3.3 Max#SAT

A first generalisation is the Max#SAT problem [FRS17], which is the simplest combinatorial optimization problem based on model counting. This problem has many applications, especially in planning [LGM98], program synthesis [FRS17] and even attack synthesis [Sah+21]. We will also show in Chapter 6 that this problem is of key importance when synthesizing non-adaptive attacks.

Definition 3.3 (Maximum model counting)**.** Given a propositional formula $\phi(X, Y, Z)$, the *maximum model counting* (Max#SAT) query:

$$\mathsf{M}X.\ \reflectbox{R}Y.\ \exists Z.\ \phi$$

asks to find the assignment x_m for variables in X such that:

$$|\mathcal{M}_Y(\phi(x_m))| = \max_{x:X\to\mathbb{B}} |\mathcal{M}_Y(\phi(x))|$$

We classify existing methods in two different categories: 1. exact resolution methods 2. approximate resolution methods. Exact methods solve (as their name implies) the problem in an exact way. Approximate resolution methods provide faster algorithm to give an approximation of the answer, note that this answer may or may not have guarantees on its correctness.

Remark 3.3.1 (Complexity)*.* Max#SAT is an $\mathsf{NP}^{\#\cdot\mathsf{NP}}$-complete problem [Mon22; Tor91], meaning that (under reasonable assumptions) the problem cannot be solved exactly without making an exponential number of calls to a model counting oracle. This in turns means that any exact resolution method would suffer from this limitation.

The use of *approximate* resolution methods can thus be helpful for resolution in practice, circumventing the complexity limitations at the expense of the exactness of the answer. Depending on the application domain, this can be a good tradeoff.

3.3.1 Exact resolution methods

`d4Max` and tree search

A first resolution method of the problem was proposed by Audemard et al. [Aud+22]. The algorithm leverages so-called *knowledge compilation* techniques in order to simplify the resolution. The idea is to leverage two key components:

1. dynamic decomposition of the formula
2. component caching

Dynamic decomposition essentially amounts to realizing that when the formula can be split in two parts that share no common variables, one can simply resolve each sub-formula independently. Component caching on the other hand is a memoization technique. One detects whether a given sub-formula was already encountered during resolution and returns the previously computed value, avoiding duplication of the computation.

The algorithm then takes *decisions* on variables, first on variables in X and then on variables on Y, checking whether the formula is satisfiable at each step, and applying the optimisations mentioned above.

An extension of this technique is then defined in order to support *weighted* MAX#SAT which is an extension of MAX#SAT allowing weights on counting variables. Note that it is possible to reduce from weighted to unweighted [Cha+15].

`erSSAT` and clause selection

$\mathsf{M}\mathsf{Я}$-SSAT [LWJ18] is a restriction of the MAX#SAT problem with empty sets of projected variables, that is $\mathsf{M}X.\ \mathsf{Я}Y.\ \phi(X, Y)$. The idea of the approach is to apply a technique called *clause selection* during the resolution process.

The technique used in this resolution method stems from the following property of $\mathsf{M}\mathsf{Я}$-SSAT:

Property 3.1 ($\mathsf{M}\mathsf{Я}$-SSAT monotony). *Given an* $\mathsf{M}\mathsf{Я}$*-SSAT problems* $\Phi = \mathsf{M}X.\ \mathsf{Я}Y.\ \phi$*:*

$$\phi(x_1) \models \phi(x_2) \implies |\mathcal{M}_Y(\phi(x_1))| \leq |\mathcal{M}_Y(\phi(x_2))|$$

Given an $\mathsf{M}\mathsf{Я}$-SSAT problem $\mathsf{M}X.\ \mathsf{Я}Y.\ \phi(X, Y)$, we can then observe the following: whenever two witnesses (i.e. assignments over X) are such that $\phi(x_1) \models \phi(x_2)$, we know that x_1 will not be more optimal than x_2.

Lee, Wang and Jiang devise a method to efficiently check for the implication mentioned above using a technique called *clause containment* [LWJ18]. Because the formulas are written in conjunctive normal form, it is sufficient to check that the remaining clauses (i.e. those that are not already satisfied) of a given witness are not contained in one of the sets of clauses of previously encountered solutions.

This can be done by associating a new literal to each clause, and progressively building a new formula (aside of the original formula) to filter assignments that are *contained* in terms of clauses by previously encountered assignments. When generating a new witness, one then wants to find a model of both formulas, ensuring that this model is *incomparable* in terms of clause set containment.

In Chapter 7 we will introduce a new algorithm based for exact resolution. This algorithm will exploit so-called *Counter-Example Guided Abstraction Refinement* techniques to speed-up the computation of the optimal answer. Furthermore, we will include other SAT resolution techniques (like symmetry breaking and equivalent literal handling) as well as provide extensions of our algorithm for *approximate resolution*.

3.3.2 Approximate resolution methods

The two algorithms presented in this section are different in nature. The first one, based on knowledge compilation, is approximate in that the answer can be shown to be non-optimal in some cases. However, the algorithm relies on “exact” and deterministic computations. The second algorithm however is a *probabilistic* algorithm, and provides guarantees.

`popcon` and knowledge compilation

As discussed in Section 3.1.4, Girol introduces, together with a definition for quantitative robustness, an algorithm to solve the Max#SAT problem [Gir22]. The algorithm proceeds in two steps:

1. compile the formula into *normal form*
2. leverage this normal form in order to solve the problem

. This approach is wide-spread in the field of *knowledge compilation*. The idea is to construct normal forms such that the computational complexity of the desired query is smaller [DM11]

In this case, the normal form used is *layered decision decomposable negation normal form* (that we will abbreviate ld-DNNF). Converse to *flat* normal forms (like CNF and DNF), decision-DNNF represent formulas as directed acyclic graphs with three types of nodes:

- $\top$ and $\bot$, representing boolean values true and false
- $\text{and}(\phi_1(V_1), \dots, \phi_n(V_n))$, with the constraints that they are *decomposable*. This means that we have $V_i \cap V_j = \emptyset$ and all i and j and ϕ_k is decision-DNNF for all k
- $\text{ite}(v, \phi(V), \psi(W))$, such that $v \notin V$ and $v \notin W$, and ϕ and ψ are in decision-DNNF

In order to get a *layered* decision-DNNF, one has to add a new constraint on *ite* nodes. Given a sequence $(A_1, \dots, A_n)$ of disjoint sets of variables, a decision-DNNF is said to be $(A_1, \dots, A_n)$-layered if for all $\text{ite}(v, \phi(V), \psi(W))$, we have $v \in A_i \implies V \cup W \subseteq \bigcup_{j=i}^{n} A_j$. Intuitively this corresponds to having decision nodes consider variables *in order*, following the order defined by the provided sets.

Conveniently, let $\phi(X, Y)$ be a (X, Y)-layered decision-DNNF formula, then computing the probability of $\mathsf{M}X.\ \mathsf{R}Y.\ \phi(X, Y)$ can be done inductively on the structure of the formula as follows:

$$\mathbf{Prob}(\text{ite}(v, \phi, \psi)) = \begin{cases} \max\{\mathbf{Prob}(\phi), \mathbf{Prob}(\psi)\} & \text{if } v \in X \\ \mathbf{Prob}(\phi) + \mathbf{Prob}(\psi) & \text{otherwise} \end{cases} \qquad \mathbf{Prob}(\top) = 1$$

$$\mathbf{Prob}(\text{and}(\phi_1, \dots, \phi_n)) = \prod \mathbf{Prob}(\phi_i) \qquad \mathbf{Prob}(\bot) = 0$$

Remark 3.3.2. Note that this method of computation can be done in time linear in the number of nodes in the representation of the formula. This in turns implies that putting the formula in such a normal form may be very costly, or lead to an exponential blowup in the size of the formula.

Furthermore, given a Max#SAT query $\mathsf{M}X.\ \mathsf{R}Y.\ \exists Z.\ \phi(X, Y, Z)$, putting ϕ into (X, Y, Z)-layered decision-DNNF allows devising a similar algorithm to solve Max#SAT.

The proposed algorithm builds on this ld-DNNF construction and provides alternative algorithms to relax the layering constraint while still providing correctness guarantees on the result. The idea is to allow the compilation step to add variables in the top level layer (effectively building a $(X \cup R, Y \setminus R)$-layered decision-DNNF) in order to cope with the complexity of the construction.

A major limitation of this algorithm is relying on knowledge compilation, and especially construction of a layered decision-DNNF. Construction of decision-DNNF [LM17] has high complexity, and especially, many queries on this normal form are linear in the size of the representation. This is turns implies that the first step of the compilation (construction of the normal form) might take a lot of time, or even not fit in memory.

`maxcount` and the Stockmeyer construction

Along with the definition of the Max#SAT problem, Fremont, Rabe and Seshia devised a probabilistic algorithm to approximate the solution. This algorithm leverages multiple key components:

- approximate model counting [MA20]
- model sampling of boolean formulas [SGM20]
- amplification thanks to Stockmeyer's technique [Sto83]

The algorithm is based on the following observation: given a Max#SAT query $\mathsf{M}X.\, ЯY.\, \exists Z.\, \phi(X, Y, Z)$, when sampling models over (X, Y) at random, the probability to sample a given model over X is related to its number of models over Y. Thus, if the optimal witness is *sufficiently far* from other witness in terms of number of models over Y, then the probability to find it by sampling a sufficient number of witnesses is high.

Conveniently, Stockmeyer provides a method to *amplify* the difference between in the distribution of the witness, using a technique called *self-product*. The technique amounts to computing the formula $\phi^k(X, \overline{Y}, \overline{Z}) \triangleq \bigwedge_{i=1}^{k} \phi(X, Y_i, Z_i)$ where Y_i and Z_i are fresh copies of the sets of variables.. Then observe that given a witness x, we have $\left|\mathcal{M}_{\overline{Y}}(\phi^k(x))\right| = |\mathcal{M}_Y(\phi(x))|^k$. Thus given a sufficient k, one can amplify the difference between the optimal witness and the other witnesses arbitrarily (by property of the exponential).

The algorithm thus relies on this fact, and provides a way to compute k as well as the number of witnesses to sample in order to provide *probably approximately correct* guarantees on the result. This means that the answer x_m returned by the algorithm follows, if we denote $\tilde{x}$ as the theoretical optimal answer:

$$\mathbb{P}\left[|\mathcal{M}_Y(\phi(x_m))| \geq \frac{|\mathcal{M}_Y(\phi(\tilde{x}))|}{1+\epsilon}\right] \geq 1 - \delta$$

Informally, this relation provides two parameters: ϵ is the *tolerance* parameter, configuring how far the answer can be from the optimal answer; δ is the *confidence* parameter, controlling to likelihood that the answer falls outside of the confidence interval. Using these two parameters, one can get arbitrarily close to the optimal answer with an arbitrary confidence in the result.

The parameters ϵ and δ can be set by the user depending on the guarantees needed on the result. The authors also provide a relaxation of the algorithm, which allows faster solving time in practice but does not provide correctness guarantees. They provide experimental evidences that the answers returned by the relaxed resolution method are actually close to being optimal.

The main limitation of this approach is the construction of the self-product. In order to provide good guarantees about the optimality of the answer, it may be needed for k to be big, in turn making the resulting formula hard to fit in memory. Assuming that the formula fits in memory, it is important to realize *solvers are not oracles*, that is, their running time depends on the size of their inputs. Thus, the running time of the sampling and counting oracles grow in the size of the formula after self-product.

In Chapter 7 we define an approximate algorithm that avoids this blowup in the size of the formula by relying on more calls to the #SAT oracle. We show experimentally that this trade-off is good in practice (Chapter 8).

3.4 SSAT

We can continue the quantitative generalisation of the SAT problem by alternating M and $\mathsf{Я}$ quantifiers. This construction effectively build what is called the *counting hierarchy*, a hierarchy of problems originally defined by Torán [Tor91]. Each level corresponds to a given quantifier prefix ($\mathsf{M}/\mathsf{Я}$ alternation), which we can describe using a *stochastic boolean problem* [Pap85]. This problem is used to synthesize adaptive attacker strategies, as we will show in Chapter 9.

Definition 3.4 (Stochastic boolean satisfiability)**.** Given a boolean formula $\phi(x_1, \ldots, x_n)$ and $Q_1, \ldots, Q_n \in \{\mathsf{M}, \mathsf{Я}\}$, the *stochastic satisfiability* (SSAT) query

$$\Phi = Q_1 x_1. \ \ldots Q_n x_n. \ \phi$$

asks to compute the *probability* $\mathbf{Prob}(\Phi)$ of the query, defined inductively on the prefix as follows:

$$\mathbf{Prob}(\mathsf{Я}y.\ \Phi) = \frac{1}{2}\mathbf{Prob}(\Phi[y \leftarrow \top]) + \frac{1}{2}\mathbf{Prob}(\Phi[y \leftarrow \bot]) \qquad \mathbf{Prob}(\top) = 1$$
$$\mathbf{Prob}(\mathsf{M}x.\ \Phi) = \max(\mathbf{Prob}(\Phi[x \leftarrow \top]), \mathbf{Prob}(\Phi[x \leftarrow \bot])) \qquad \mathbf{Prob}(\bot) = 0$$

Remark 3.4.1. The provided definition is slightly different from the standard one from [Pap85]. This is done to make the emphasis on model counting and to simplify the statement of further results during this thesis.

The “standard” definition allows adding weights on counting quantifiers, slightly altering the underlying problem, but keeping the same computational complexity. It is possible though to transform a *weighted* SSAT query into a unweighted one without exponential blowup, assuming that the weights are rational [Cha+15].

Contrary to Max#SAT, there are no approximate resolution methods for SSAT at the time of writing. We will thus only mention exact resolution methods.

`DC-SSAT` and `DPLL` The `DPLL` [DP60] is a standard technique to check for the satisfiability of a boolean formula in conjunctive normal form. It is based on *resolution*, which is the following proof rule

$$\frac{l \vee c_1 \qquad \neg l \vee c_2}{c_1 \vee c_2}$$

Interestingly, generalisations of the resolution rule can be applied in SSAT solving [TF10], allowing to compute the probability of the formula. `DC-SSAT` [MB05] is an early implementation of this technique to SSAT solving. The idea is to split the formula into sub-problems based on the prefix as well as the clause set itself. One can then solve each sub-problem independently, applying techniques like resolution, and reconstructing the optimal answer later on.

`ClauSSat` and clause selection As mentioned before, clause selection is a well known technique in SAT solving that can has been successfully applied to MAX#SAT [LWJ18]. This technique can be lifted to SSAT in a similar fashion as discussed previously. This is the basis of the `ClauSSat` algorithm [CHJ21].

The algorithm is split in two mutually recursive functions: 1. handling Я quantified levels of the prefix 2. handling M quantified levels of the prefix. Solving the complete SSAT problem thus amounts to calling the appropriate function on the top level of the prefix.

Regarding maximizing quantified levels, the algorithm is very similar to the one for MЯ-SSAT [LWJ18] (see Section 3.3.1).

Regarding counting quantified levels, we can also apply clause selection in this setting. Note first that the problem of solving counting quantified levels is a problem called *discrete integration*, meaning that one has to compute the sum of a function over a discrete (and exponential in size) set. The idea is to rely on a generalisation of Property 3.1 to SSAT. This means that in order to compute the discrete integration, one only has to iterate over the possible clause sets instead of the possible assignments.

One though still has to account for the fact that multiple assignments may lead to a single clause selection. This can be accounted for by a call to a model counter to aggregate the results of the current Я quantified level.

`SharpSSAT` and component caching `SharpSSAT` [FJ23] applies well-known techniques used in model counting. It is essentially the lifting of dynamic decomposition and component caching as used by `d4max`.

The algorithm differs from other SSAT solvers by reconstructing the optimal witness associated with the probability computation. This is done by following the trace of execution of the algorithm and building the witness along it. The paper [FJ23] also extends `ClauSSat` with witness extraction capabilities in order to compare the two methods.

In Chapter 9, we will show the SSAT is a good formalism to encode the problem of synthesis of optimal strategies for adaptive attackers. We will also perform an experimental evaluation of said encoding.

3.5 DSSAT

In this thesis, we will mention one more step in terms of expressiveness in the formulas encountered. In order to take this step, we use so-called *Henkin quantifiers* [HK65]. The idea is to relax the requirements imposed by quantifier alternation, and allow arbitrary dependencies between variables. This defined the DSSAT problem. DSSAT is a newer problem, and as such has less resolution methods at the time of writing. To our knowledge, only one exists and is not implemented in practice.

Definition 3.5 (Dependency stochastic boolean satisfiability)**.** Given a boolean formula $\phi(Y, x_1, \ldots, x_n)$ and $H_1, \ldots, H_n \subseteq Y$, the *dependency stochastic satisfiability* (DSSAT) query

$$\mathsf{Я}Y.\ \mathsf{M}^{H_1} x_1.\ \ldots \mathsf{M}^{H_n} x_n.\ \phi$$

asks to find a substitution $\sigma : X \to \mathcal{F}\langle Y \rangle$ such that:

- for all x_i, the substitution respects the dependencies, that is $\sigma(x_i) \in \mathcal{F}\langle H_i \rangle$
- the number of models of $\phi[x_i \leftarrow \sigma(x_i)]$ is maximal

Remark 3.5.1. Similarly to Remark 3.4.1, we provide a definition of DSSAT that puts the emphasis on model counting. The standard definition allows for weights for each counting variable, but it is possible to turn a weighted query into an unweighted one [Cha+15].

Luo, Cheng and Jiang [LCJ23] also propose a resolution proof system. Developing a proof system for DSSAT allows mainly the ease the verification of DSSAT solvers (which do not exist yet) and it would allow the verification of the proof generated by the solver[2].

The authors continue and show that the proof system proposed is sound and complete, meaning that it is a fitting proof system for DSSAT and a good verification tool for future solvers.

Furthermore, this calculus allows the construction of the witness associated with the probability computation, and thus allows, from the original proof of the computation of the probability, to reconstruct the maximizing witness.

We will show in Chapter 11 that a version of this problem can be used to synthesize attack strategies for multiple attackers, the last step in the attacher hierarchy. In Chapter 13 we will present alternative algorithms to solve the DSSAT problem. This is motivated by resolution in practice, and implemented in a prototype tool.

[2]This is a standard technique already applied in SAT solving. The solver generates a proof in the DRAT format, which can then be checked by a proof checker. Note that (under reasonable assumptions) checking the proof is much easier than generating it, and especially one can easily build a formally verified checker.

Part II

Programs and attacks

Chapter 4

The SIP language

We define in this chapter the SIP *formal language* destined to serve as an intermediate step when analyzing programs. This language can be viewed as a framework between actual security evaluation and counting problems and aims to be expressive enough to allow to concisely express *source program* behaviors but also simple enough to allow simple definitions for security problems.

We chose a simple language that makes input/output handling explicit, as well as with a strict execution model. This effectively turns our language into a concise representation for executions in the source program that reach the point of interest. This means that one needs to first analyze said program to search for paths reaching the target point and then represent them into a program in SIP, which will allow answering questions about how easy an attacker can exploit the selected set of paths to reach the vulnerability.

Absence of loops It is important to note that SIP has no loop construct. Based on the comments made above, this is because it is deemed sufficient to only consider a (finite) subset of (finite) paths within the program to see if they contain a problem. This is the classical security/safety debate where security (resp. safety) is about finding bugs (resp. proving their absence). Furthermore, when we consider a fully-fledged programming language (with loops), one is quickly confronted to undecidability of most questions and non-termination.

Explicit interactions with the environment SIP contains two constructs to handle interactions with the program's environment: one for inputting values to the program, the other for the program to output values. Note that this enforces an explicit handling of all the informations given by the program with regard to its execution. For instance, side-channels *must be made* explicit within the formal representation of the program in order for them to be usable within our framework.

Absence of random sampling While programs are allowed to access randomly generated values, it is important to note that SIP programs do not have the ability to sample new random values during the program's run. This restriction is not very important as (for finite executions) the randomness is modelled by the sampling of the initial state of the program.

Expression	e	::=	x	*program variable in X*
		|	y	*random variable in Y*
		|	v	*value in* **Val**
		|	`ite`(e, e, e)	*if then else*
		|	$\star_1\ e$	*unary operation*
		|	$e \star_2 e$	*binary operation*
Program	P	::=	x `<-` e	*assignment*
		|	x `<- in()`	*input*
		|	`out(`x`)`	*output*
		|	`skip`	*no operation*
		|	`if` e `then` P `else` P `end`	*conditional statement*
		|	P `;` P	*sequence*

Figure 4.1: SIP grammar

4.1 SIP syntax

We define the *simple interactive* programming language (SIP) as the language described by the grammar in Figure 4.1.

We distinguish two disjoint finite sets of variables: X is the set of program variables that can be assigned within the program and Y is the set of random variables, whose value can only be read but never changed by the program (as per the syntax definition).

Variables are assumed to range over a finite domain denoted by **Val**, which is equipped with a set of binary (resp. unary) operations denoted by $\star_2$ (resp. $\star_1$) in the syntax. These operations are assumed to be total, meaning that the evaluation of any operator on any value will always yield a new value in the **Val**, and never error. We also assume that the domain contains two boolean values $\top$ and $\bot$ denoting boolean values true and false along with the classical boolean logic operators.

We finally assume that expressions that serve as conditions for `if` statements and `ite` expressions evaluate to one of the two boolean values. This in turns ensure that the programs always terminate and never get stuck: it is not possible by assumption on the structure of programs to perform undefined operations within `if` statements and `ite` expressions.

Definition 4.1 (Variables of a program)**.** Given a SIP program P we define the following:

- $\mathbf{Vars}(P) = X \cup Y$ the variable occurring in P
- $\mathbf{Out}(P) \subseteq X$ the variables occurring in `out` statements
- $\mathbf{In}(P) \subseteq X$ the variables occurring in `in` statements
- $\mathbf{Rand}(P) = Y$ the random variables occurring in P

Example 4.1.1. Let $\mathbf{Val} = \{1, 2, ..., N\}$, assuming N is even, we assume that the domain is equipped with the *saturating* addition operation, that is:

$$x \oplus y = \begin{cases} x + y & \text{if } x + y \leq N \\ N & \text{otherwise} \end{cases}$$

The following is a valid SIP program:

```
z <- y1 ⊕ y2
out(z)
x <- in()
```

We also have:

$$\begin{aligned} \textbf{Vars}(P) &= \{x, y_1, y_2, z\} \\ \textbf{Out}(P) &= \{z\} \\ \textbf{In}(P) &= \{x\} \\ \textbf{Rand}(P) &= \{y_1, y_2\} \end{aligned}$$

4.2 Standalone SIP semantics

We define operational semantics for SIP programs. This semantic defines the small steps between program states, with event on the transitions. We for now restrict ourselves to the semantics without taking into account the attack against which the program runs, which we will define later.

Definition 4.2 (States)**.** A state σ is a function from variables to values, that is:

$$\sigma : X \cup Y \to \textbf{Val}$$

We denote by **State** the set of all states. We assume a natural lifting of states to expressions such that for any expression e, $\sigma(e) \in \textbf{Val}$.

Finally we denote the overloading of a value of a state in the usual way, with $v \in \textbf{Val}$ and $u \in X \cup Y$:

$$\sigma[u \leftarrow v](w) \triangleq \begin{cases} v & \text{if } u = w \\ \sigma(w) & \text{otherwise} \end{cases}$$

Definition 4.3 (Event)**.** An event is a tagged value, taking the form of two constructors:

- in(v) denotes input events
- out(v) denotes output events

Definition 4.4 (Semantics)**.** We define the semantics of SIP programs as a labeled transition system from SIP $\times$ **State** to (SIP $\cup \{\bullet\}) \times$ **State**. We use $\bullet$ as the marker for the end of the execution of a program.

Let P and P' be two programs, and let σ and σ' be two program states, we have the following kinds of transitions:

$$\frac{}{(\texttt{out}(x), \sigma) \xrightarrow{\text{out}(\sigma(x))} (\bullet, \sigma)}$$

(a) Output

$$\frac{v \in \mathbf{Val}}{(x \texttt{ <- } \texttt{in}(), \sigma) \xrightarrow{\text{in}(v)} (\bullet, \sigma[x \leftarrow v])}$$

(b) Input

$$\frac{\sigma(e) = \top}{(\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end}, \sigma) \rightarrow (P_1, \sigma)} \quad \frac{\sigma(e) = \bot}{(\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end}, \sigma) \rightarrow (P_2, \sigma)}$$

(c) Conditional statement

$$\frac{}{(x \texttt{ <- } e, \sigma) \rightarrow (\bullet, \sigma[x \leftarrow \sigma(e)])}$$

(d) Assignment

$$\frac{}{(\texttt{skip}, \sigma) \rightarrow (\bullet, \sigma)}$$

(e) Skip

$$\frac{(P_1, \sigma) \xrightarrow{E} (P_1', \sigma')}{(P_1 \ ; \ P_2, \ \sigma) \xrightarrow{E} (P_1' \ ; \ P_2, \ \sigma')} \quad \frac{(P_1, \sigma) \xrightarrow{E} (\bullet, \sigma')}{(P_1 \ ; \ P_2, \ \sigma) \xrightarrow{E} (P_2, \ \sigma')}$$

(f) Sequence

Figure 4.2: Standalone semantics for SIP

- Silent transition: $(P, \sigma) \rightarrow (P', \sigma')$
- Transition outputting a value v: $(P, \sigma) \xrightarrow{\text{out}(v)} (P', \sigma')$
- Transition under input value v: $(P, \sigma) \xrightarrow{\text{in}(v)} (P', \sigma')$

The rules for the transition system are described in Figure 4.2.

Rules “Output” (Figure 4.2a) and “Input” (Figure 4.2b) are concerned with the handling of the communication between the SIP program being executed and the environment. In the case of “Input”, the execution proceeds by assigning to the target variable the value gathered from the environment, firing an “in” event. Dually, the “Output” rule reads the value of a variable in the current execution state and outputs it to the environment. Rules “Assignment” (Figure 4.2d) and “Skip” (Figure 4.2e) are the standard rules to describe the evaluation of an expression and assignment to a variable, as well as no operation. In Figures 4.2c and 4.2f one can find the classical rules for execution of compound programs. The case of sequence is split in two as we do not restrict our sequence operation to only statements, but sequence of arbitrary programs.

Remark 4.2.1. Semantics described in Definition 4.4 are non-deterministic. This is because the values that the program can receive in `input` statements have no constraints beyond being in the **Val** set.

Example 4.1.1 (continuing from p. 36). Let σ_0 be the following initial state:

$$\sigma_0(x) = \{(y_1, 1), (y_2, 2), (z, 0), (x, 0)\}$$

The following transitions are all valid:

$$\begin{aligned}
&(z \texttt{ <- } y_1 \oplus y_2 \texttt{ ; } \texttt{out}(z) \texttt{ ; } x \texttt{ <- } \texttt{in}(), \sigma_0)\\
\rightarrow&(\texttt{out}(z) \texttt{ ; } x \texttt{ <- } \texttt{in}(), \{(y_1,1),(y_2,2),(z,3),(x,0)\})\\
\xrightarrow{\text{out}(3)}&(x \texttt{ <- } \texttt{in}(), \{(y_1,1),(y_2,2),(z,3),(x,0)\})\\
\xrightarrow{\text{in}(4)}&(\bullet, \{(y_1,1),(y_2,2),(z,3),(x,4)\})
\end{aligned}$$

4.3 SIP semantics under attack

To define the semantics of SIP programs under attack we need to track the information exchange as the program executes. This is done by accumulating the sequence of events that the program goes through as it executes.

Definition 4.5 (Traces)**.** A *trace* is a sequence of events. We denote by **Trace** the set of all traces.

Given a trace τ we also define the following:

- $\tau\downharpoonright_{\text{in}}$ is the sequence of input events within τ
- $\tau\downharpoonright_{\text{out}}$ is the sequence of output events within τ
- τ_i denotes the i-th element of τ
- $\tau_{<i}$ is the trace *up to* the i-th element in the trace.

Example 4.3.1. The following is a trace:

$$\tau \triangleq [\text{out}(1), \text{in}(2), \text{out}(3)]$$

And we have:

$$\begin{aligned}
\tau\downharpoonright_{\text{in}} &= [\text{in}(2)] & \tau\downharpoonright_{\text{out}} &= [\text{out}(1), \text{out}(3)]\\
\tau_{<2} &= [\text{out}(1)] & \tau_3 &= \text{out}(3)
\end{aligned}$$

In the following we mean by *configuration* elements of the set **State** × **Trace**. Note that we can view the semantics under attack as a transition system over configuration.

We can now define what attacks are. Conceptually, an attack is just a way to compute, given what the interactions with the program were thus far, the next input value. We do this by representing attacks as (total) functions, that take a trace as input and return a value as output.

Definition 4.6 (Attack)**.** An *attack* $\mathcal{A}$ is a total function from traces to values, that is:

$$\mathcal{A} : \textbf{Trace} \rightarrow \textbf{Val}$$

In this thesis we are only interested in deterministic attacks, thus justifying the choice of a function to represent the attack being carried.

We distinguish here the *attack* (the steps that one has to take to reach a goal), and the *attacker* which is the entity whose goal is to devise an attack against the program. The effectiveness of a given attack is defined with regard to the attacker's *goal*, a notion we discuss later in this chapter, as some preliminary definitions are necessary.

We now define the semantics *under attack*, that is the semantics when the program is actually running against another player (the attack). This new version of the semantics is parametrized by the attack.

Definition 4.7 (Semantics under attack)**.** We define the semantics under attack for SIP programs as a transition system from $\mathsf{SIP} \times \mathbf{State} \times \mathbf{Trace}$ to $(\mathsf{SIP} \cup \{\bullet\}) \times \mathbf{State} \times \mathbf{Trace}$, parametrized by an attack $\mathcal{A}$.

The complete definition of the transition system is defined in Figure 4.3. We denote the transitive closure of this relation as $(P, \sigma, \tau) \underset{\mathcal{A}}{\rightarrow} * (P', \sigma', \tau')$.

Note that while the standalone semantics is a labeled transition system, the semantics under attacks is not labeled. The events present along the execution of a program in the standalone semantics appear as part of the configuration in the semantics under attack. This is needed to represent the evolution of the *knowledge* accumulated by the attack as the program executes.

Furthermore, the rule for "Input" (Figure 4.3b) is now changed as the program now executes against the attack. The value that the program gets for the environment is computed by the attack using the trace of events that occurred previously during the execution.

Note that given a program P and an attack $\mathcal{A}$, the pair $(P, \mathcal{A})$ actually defines a deterministic transition system. That is, once the initial values for the variables are chosen, the entire execution is performed without resorting to any kind of randomness whatsoever.

Lemma 4.1. *Given a* SIP *program P and $(\sigma, \tau) \in \mathbf{State} \times \mathbf{Trace}$, there exist a unique pair (σ', τ') such that:*

$$(P, \sigma, \tau) \underset{\mathcal{A}}{\rightarrow} * (\bullet, \sigma', \tau')$$

Moreover:

$$\forall y \in \mathbf{Rand}(P).\ \sigma(y) = \sigma'(y)$$

Proof. Existence is a consequence of the fact that execution cannot be stuck. There are only two rules in the semantics with premises (other than induction rules):

- "Conditional statement" (Figure 4.3c): in this case, by hypothesis on programs, the condition in `if` statements can only evaluate to one of the boolean values. This ensure that either one of the rules applies.
- "Input" (Figure 4.3b): in this case, because the function $\mathcal{A}$ is total, it is guaranteed to return a value.

Unicity is a consequence of the fact that $\cdot \underset{\mathcal{A}}{\rightarrow} \cdot$ is a deterministic relation: there cannot be two rules that apply at the same time.

Finally, the fact that variables in $\mathbf{Rand}(P)$ do not change value is a consequence of the syntax of programs: variables in $\mathbf{Rand}(P)$ cannot appear as the left-hand side operands in assign statements, and thus they cannot change value during the execution. $\square$

$$\frac{}{(\texttt{out}(x), \sigma, \tau) \xrightarrow[\mathcal{A}]{} (\bullet, \sigma, \tau \cdot \mathrm{out}(\sigma(x)))}$$

(a) Output

$$\frac{v = \mathcal{A}(\tau)}{(x \texttt{ <- } \texttt{in}(), \sigma, \tau) \xrightarrow[\mathcal{A}]{} (\bullet, \sigma[x \leftarrow v], \tau \cdot \mathrm{in}(v))}$$

(b) Input

$$\frac{\sigma(e) = \top}{(\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end}, \sigma, \tau) \xrightarrow[\mathcal{A}]{} (P_1, \sigma, \tau)} \qquad \frac{\sigma(e) = \bot}{(\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end}, \sigma, \tau) \xrightarrow[\mathcal{A}]{} (P_2, \sigma, \tau)}$$

(c) Conditional statement

$$\frac{}{(x \texttt{ <- } e, \sigma, \tau) \xrightarrow[\mathcal{A}]{} (\bullet, \sigma[x \leftarrow \sigma(e)], \tau)}$$

(d) Assignment

$$\frac{}{(\texttt{skip}, \sigma, \tau) \xrightarrow[\mathcal{A}]{} (\bullet, \sigma, \tau)}$$

(e) Skip

$$\frac{(P_1, \sigma, \tau) \xrightarrow[\mathcal{A}]{} (P_1', \sigma', \tau')}{(P_1 \texttt{ ; } P_2, \sigma, \tau) \xrightarrow[\mathcal{A}]{} (P_1' \texttt{ ; } P_2, \sigma', \tau')} \qquad \frac{(P_1, \sigma, \tau) \xrightarrow[\mathcal{A}]{} (\bullet, \sigma', \tau')}{(P_1 \texttt{ ; } P_2, \sigma, \tau) \xrightarrow[\mathcal{A}]{} (P_2, \sigma', \tau')}$$

(f) Sequence

Figure 4.3: SIP semantics under attack

Definition 4.8 (Final state)**.** Given a SIP program P and $(\sigma, \tau) \in \textbf{State} \times \textbf{Trace}$, we define the *final state* of P when starting from configuration (σ, τ) as the unique pair (σ', τ') such that:

$$(P, \sigma, \tau) \xrightarrow[\mathcal{A}]{}{*}\ (\bullet, \sigma', \tau')$$

We denote this pair as $[\![P]\!]_{\mathcal{A}}(\sigma, \tau) \triangleq (\sigma', \tau')$.

4.4 Effectiveness of attacks

We provide in this section a few definition of *effectiveness* of an attack that are based on probability theory. The idea is to use definition that are straightforward to define from the security point of view, and that effectively capture well know attacker objectives.

4.4.1 Programs as distribution transformers

As the combination of a program and an attack is deterministic, we can define a probabilistic version of the semantics of programs as the lifting of the semantics on individual configurations to distributions of configurations. We thus lift the definition of final states (Definition 4.8) to distributions over configurations as the lifting of the $[\![P]\!]_{\mathcal{A}}(\cdot)$ function from individual configurations to distributions over configurations.

Definition 4.9 (Distribution transformers)**.** Given a SIP program P, and $\delta \in \mathbb{D}(\textbf{State} \times \textbf{Trace})$, we define $[\![P]\!]_{\mathcal{A}}(\delta)$ as the point-wise lifting of Definition 4.8 to distributions.

$$\begin{tikzcd}\mathbb{D}(\textbf{State} \times \textbf{Trace}) \arrow[r] & \boxed{[\![P]\!]_{\mathcal{A}}} \arrow[r] & \mathbb{D}(\textbf{State} \times \textbf{Trace})\end{tikzcd}$$

Figure 4.4: Schematic vision for programs

Furthermore, we define as an abuse of notation the *posterior* distribution on states as the projection of $[\![P]\!]_{\mathcal{A}}(\delta)$ on to the set of states: $[\![P]\!]_{\mathcal{A}}\downharpoonleft_{\textbf{State}}(\delta) \in \mathbb{D}(\textbf{State})$. Similarly, we define the distribution of possible traces of P as $[\![P]\!]_{\mathcal{A}}\downharpoonleft_{\textbf{Trace}}(\delta) \in \mathbb{D}(\textbf{Trace})$.

It is clear from here that a program together with an attack can be viewed as a distribution transformer. Note though that this transformation is *deterministic*, just like the semantics under attack, which means that it maps any distribution over configurations to a unique final distribution and not to a distribution over distributions (which is called a *hyper-distribution*).

We introduce in Figure 4.4 a schematic notation for programs that presents them as distribution transformers, and that will help define further notions.

Now that we have described programs as distribution transformers, we can now define security measures as properties of these distribution transformers. The remainder of this chapter is dedicated to that, starting by special distributions over configurations that will allow us to define properly the security measures of interest.

Definition 4.10 (Initial states)**.** We deem as *initial states* the states where all program variables in X are set to a default value $0_{\textbf{Val}}$ (for example, $\bot$ for boolean values). More formally, a state σ_0 is deemed initial whenever:

$$\forall u \in X, \sigma_0(x) = 0_{\textbf{Val}}$$

We denote the set of initial states as $\textbf{State}_0$. Note that $|\textbf{State}_0| = |\textbf{Val}|^{|\textbf{Rand}(P)|}$ because only the variables in $\textbf{Rand}(P)$ can have arbitrary values.

Definition 4.11 (Initial configurations distribution)**.** We define the *initial configurations* distribution as the uniform distribution on configurations corresponding to initial states, that is:

$$\begin{aligned}\Sigma_0 &\triangleq \mathcal{U}\big(\{(\sigma_0, []) \mid \sigma_0 \in \textbf{State}_0\}\big)\\ &= (\sigma, \tau) \mapsto \begin{cases}\dfrac{1}{|\textbf{Val}|^{|\textbf{Rand}(P)|}} & \text{if } \sigma \in \textbf{State}_0 \text{ and } \tau = []\\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

Informally, Definition 4.11 describes states where the values for the variables in $\textbf{Rand}(P)$ are sampled uniformly at random. From this, one can define objectives that the attacker may want to fulfill as properties of the distribution of the configurations after execution on the initial states distribution. We define two such objectives in what follows, as well as introducing the problem of attack synthesis in both cases.

$$\begin{tikzcd}\mathbb{D}(\mathbf{State} \times \mathbf{Trace}) \arrow[r] & \boxed{[\![P]\!]_{\mathcal{A}}\restriction_{\mathbf{State}}} \arrow[r] & \mathbb{D}(\mathbf{State}) \arrow[r] & \boxed{e} \arrow[r] & \mathbb{D}(\{\top, \bot\})\end{tikzcd}$$

Figure 4.5: Schematic of the quantitative reachability transformer

4.4.2 Quantitative reachability

Quantitative reachability essentially amounts to computing the probability that a given predicate over the final state is true. This in turns amounts to considering the program's distribution transformer together with the predicate to evaluate as one single distribution transformer, with which one wants to compute the probability of the event $\top$.

Figure 4.5 presents this using the previously used schematic notation. We can use this quantitative reachability transformer to define formally the objective an attacker may want to fulfill.

Definition 4.12 (Quantitative reachability)**.** Given a program P and an attack $\mathcal{A}$, we define the quantitative reachability of a boolean expression e over the variables of P as:

$$\mathcal{R}_P^e(\mathcal{A}) \triangleq \mathbb{P}\big[[\![P]\!]_{\mathcal{A}}\restriction_{\mathbf{State}}(\Sigma_0)(e) = \top\big]$$

Note that the quantitative reachability (as a measure) is thus defined as a property of the posterior distribution of the program when applied to the uniform distribution on initial states.

We can now formally define the problem of attack synthesis with regard to quantitative reachability.

Definition 4.13. Given a program P and a boolean expression e over the variables of P, we define the *attack synthesis* with regard to the quantitative reachability objective e as the problem of computing $\mathcal{A}^\star$ such that:

$$\mathcal{R}_P^e(\mathcal{A}^\star) = \max_{\mathcal{A}} \mathcal{R}_P^e(\mathcal{A})$$

In order to effectively compute the value of the probability in Definition 4.12, we need to simplify the expression. It is easy to see that in fact, the problem of computing the quantitative reachability of a predicate e amounts to a counting problem.

Property 4.2. *Given a program P in normal form and a boolean expression e, the following holds for any attack $\mathcal{A}$*

$$\mathcal{R}_P^e(\mathcal{A}) = \frac{|\{\sigma' \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) \mid \sigma'(e) = \top\}|}{|\mathbf{Val}|^{|\mathbf{Rand}(P)|}}$$

Proof. This is a consequence of Lemma 4.1. Indeed because the values of the variables in **Rand**(P) do not change, the number of distinct final states is exactly the number of possible values that one can assign to the variables in $\mathbf{Rand}(P)$. More formally:

$$\begin{aligned}\mathcal{R}^e_P(\mathcal{A}) &= \frac{|\{\sigma' \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) \mid \sigma'(e) = \top\}|}{|\{\sigma' \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0)\}|} \\ &= \frac{|\{\sigma' \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) \mid \sigma'(e) = \top\}|}{|\{\sigma \in \mathbf{State}_0 \mid [\![P]\!]_{\mathcal{A}}(\sigma, []) = (\sigma', \tau')\}|} \\ &= \frac{|\{\sigma' \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) \mid \sigma'(e) = \top\}|}{|\mathbf{Val}|^{|\mathbf{Rand}(P)|}}\end{aligned}$$

□

Example 4.1.1 (continuing from p. 36). Let us consider the target expression $e \triangleq y_1 \leq x \leq y_2$. We can compute the effectiveness of various attacks with regard to the quantitative reachability of e.

Recall that $\mathbf{Val} = \{1, \ldots, N\}$ with N even. Let us first consider the following attack, with $c \in \mathbf{Val}$:

$$\mathcal{A}_1 \triangleq [\mathrm{out}(\cdot)] \mapsto c$$

In this case we have:

$$\begin{aligned}\mathcal{R}^e_P(\mathcal{A}_1) &= \mathbb{P}[y_1 \leq c \leq y_2] \\ &= \mathbb{P}[y_1 \leq c] \times \mathbb{P}[c \leq y_2] \\ &= \frac{c}{N}\frac{N - c + 1}{N}\end{aligned}$$

We can see that the quantitative reachability of attacks of this form depends on the value that the attack inputs (i.e. c), with the optimal value being $c_m = \frac{N}{2}$, with a quantitative reachability of $\frac{1}{4}$.

Let us now consider the following attack:

$$\mathcal{A}_2 \triangleq [\mathrm{out}(v)] \mapsto \frac{v}{2}$$

We can compute the quantitative reachability in this case too:

$$\begin{aligned}\mathcal{R}^e_P(\mathcal{A}_2) = \mathbb{P}[y_1 \leq{}& \frac{y_1 \oplus y_2}{2} \leq y_2] \\ &= \mathbb{P}[y_1 + y_2 > N \wedge y_1 \leq \frac{N}{2} \leq y_2] + \mathbb{P}[y_1 + y_2 \leq N \wedge y_1 \leq \frac{y_1 + y_2}{2} \leq y_2] \\ &= \mathbb{P}[y_1 + y_2 > N \wedge y_1 \leq \frac{N}{2} \leq y_2] + \mathbb{P}[y_1 + y_2 \leq N \wedge y_1 \leq y_2]\end{aligned}$$

$$\begin{tikzcd}\mathbb{D}(\textbf{State} \times \textbf{Trace}) \arrow[r] & \boxed{[\![P]\!]_{\mathcal{A}}\downharpoonleft_{\textbf{Trace}}} \arrow[r] & \mathbb{D}(\textbf{Trace})\end{tikzcd}$$

Figure 4.6: Schematic of the leakage transformer

We compute each part of the sum:

$$\begin{aligned}
P_1 &= \mathbb{P}\left[y_1 + y_2 > N \wedge y_1 \le \frac{N}{2} \le y_2\right] \\
&= \mathbb{P}\left[y_1 \le \frac{N}{2} \wedge y_2 > N - y_1 \wedge y_2 \ge \frac{N}{2}\right] \\
&= \mathbb{P}[y_1 \le \frac{N}{2} \wedge y_2 > N - y_1] \\
&= \sum_{i=1}^{\frac{N}{2}} \frac{1}{N}\frac{i}{N} \\
&= \frac{1}{N^2}\frac{\frac{N}{2}\left(\frac{N}{2}+1\right)}{2} \\
&= \frac{N+2}{8N} \\
&= \frac{1}{8} + \frac{1}{4N}
\end{aligned}$$

$$\begin{aligned}
P_2 &= \mathbb{P}\left[y_1 + y_2 \le N \wedge y_1 \le \frac{y_1 + y_2}{2} \le y_2\right] \\
&= \mathbb{P}[y_1 \le y_2 \wedge y_1 \le N - y_2] \\
&= \mathbb{P}[y_1 \le \min(y_2, N - y_2)] \\
&= \begin{matrix}\mathbb{P}[y_2 \le \frac{N}{2} \wedge y_1 \le y_2] + \\ \mathbb{P}[y_2 > \frac{N}{2} \wedge y_1 \le N - y_2]\end{matrix} \\
&= \sum_{i=1}^{\frac{N}{2}} \frac{1}{N}\frac{i}{N} + \sum_{i=\frac{N}{2}+1}^{N} \frac{1}{N}\frac{N-i}{N} \\
&= \frac{1}{N^2}\frac{\frac{N}{2}\left(\frac{N}{2}+1\right)}{2} + \frac{1}{N^2}\frac{\frac{N}{2}\left(\frac{N}{2}-1\right)}{2} \\
&= \frac{1}{4}
\end{aligned}$$

This leads to:

$$\mathcal{R}^e_P(\mathcal{A}_2) = \frac{3}{8} + \frac{1}{4N} > \mathcal{R}^e_P(\mathcal{A}_1)$$

4.4.3 Leakage

The leakage objective is substantially different from the quantitative reachability in nature. The objective here is not to maximize the probability of some favorable event from the point of view of the attacker but to maximize the amount of information leaked by the program. The associated distribution transformer is presented in Figure 4.6.

This security measure quantifies how much information about the secrets of the program (that is, the random variables) is leaked through its outputs. In order to quantify the amount of information leaked, we resort to classical *quantitative information flow* notions.

We use *mutual information* as a measure of leakage in Definition 4.14. It is known [Smi09] that this measure captures the information leaked by a given program. Intuitively, this measure captures the number of *bits of information* the program outputted to the network with regard to its internal state.

Definition 4.14 (Leakage)**.** Given a program P and an attack $\mathcal{A}$, we define the leakage induced by $\mathcal{A}$ on P as:

$$\mathcal{L}_P(\mathcal{A}) \triangleq \mathbb{I}\left[[\![P]\!]_{\mathcal{A}}\downharpoonleft_{\textbf{State}}(\Sigma_0), [\![P]\!]_{\mathcal{A}}\downharpoonleft_{\textbf{Trace}}\downharpoonleft_{\text{out}}(\Sigma_0)\right]$$

Intuitively, the leakage measures how easy for the attacker it is to guess the state the program ends up in only by observing the execution trace of the program when playing an attack. Changing the attack thus may allow leaking more information.

Similarly as in Section 4.4.2, we can now define the problem of attack synthesis in the context of a leakage objective.

Definition 4.15. Given a program P, we define the problem of *attack synthesis* with regard to a leakage objective as finding $\mathcal{A}^\star$ such that:

$$\mathcal{L}_P(\mathcal{A}^\star) = \max_{\mathcal{A}} \mathcal{L}_P(\mathcal{A})$$

Unfortunately, computing the mutual information is hard in general in practice. However, it has been shown[Smi09] that for deterministic programs, and if the input distribution (i.e. Σ_0) is uniform, then one can simplify the computation of said mutual information.

Property 4.3. *Given a program P and an attack $\mathcal{A}$, we have:*

$$\mathcal{L}_P(\mathcal{A}) = \log_2 \left| [\![P]\!]_{\mathcal{A}} \restriction_{\textbf{Trace}} \restriction_{\text{out}} (\textbf{State}_0) \right|$$

Proof. This is the result shown by Smith [Smi09, Theorem 1] applied to the distribution transformer $[\![P]\!]_{\mathcal{A}}$. □

Example 4.4.1. Let P denote the following program over bounded integers (where N denotes the upper-bound):

```
x <- in()
z <- x > y
out(z)
x <- in()
z <- x > y
out(z)
```

In this case, taking $\mathcal{A}_1 \triangleq \cdot \mapsto 42$, we can see that the only two possible traces (projected on the outs) are: $[\text{out}(\top), \text{out}(\top)]$ and $[\text{out}(\bot), \text{out}(\bot)]$. It follows that:

$$\mathcal{L}_{\mathcal{A}_1}(P) = 1$$

However taking the attack $\mathcal{A}_2$ defined as follows:

$$\mathcal{A}_2 = \begin{cases} [] & \mapsto \frac{N}{2} \\ [\text{in}(\cdot), \text{out}(\top)] & \mapsto \frac{N}{4} \\ [\text{in}(\cdot), \text{out}(\bot)] & \mapsto \frac{3N}{4} \end{cases}$$

In this case, the possible traces are all the possible combinations of $\text{out}(\top)$ and $\text{out}(\bot)$ repeated twice, which yields:

$$\mathcal{L}_{\mathcal{A}_2}(P) = 2$$

Actually, $\mathcal{A}_2$ is the maximally leaking attack for P. This is because it covers all the possible values that the trace could take.

Concluding remarks

The computability of the attack function is something of interest to discuss. For example, if we consider SIP programs over integers modulo a prime p, with the traditional definition for exponentiation over this field, one could in theory write the following program:

```
z <- g ^ y
out(z)
x <- in()
```

Let us further assume that the attacker want to find the optimal attack with regard to the quantitative reachability objective $e \triangleq x = y$, which corresponds to the objective of guessing the value of y.

An optimal attack exists indeed, as it is theoretically possible to invert the exponentiation operation: this is the so-called discrete logarithm problem over finite fields. However, the function computing the correct x as per the quantitative reachability objective mentioned above would run in exponential time with regard to the number of digits of p.

This observation thus questions the relation (in terms of computational complexity) between the optimal attack function, and the process with which one comes up with said optimal attack.

We will see in the remainder of this thesis, that actually, the language in which one wants to write the attack function influences greatly both the complexity of the synthesis of the attack, and the complexity class that a given attack function is allowed to be in. Namely, in Part III we discuss the case of attack functions being constants, and in Part IV we discuss the case of attack functions being boolean functions.

Chapter 5

Normalizing SIP programs

In this chapter, our goal is to transform SIP programs towards a normal form which preserves some properties of the original program, while limiting and simplifying their study. The various transformations involved are discussed one by one, and this will allow to only consider programs in normal form in the remainder of this thesis.

We begin by a notion of program equivalence that is of importance in this chapter. Our objective will be to prove that the following equivalence relation holds through our program transformations, and that this program equivalence entails a preservation of both the quantitative reachability and the leakage of the programs.

5.1 Program equivalence

Definition 5.1 (E-equality for configurations)**.** We say that two configurations $(\sigma_1, \tau_1), (\sigma_2, \tau_2)$ are *equal on E* whenever:

1. $\tau_1 = \tau_2$
2. $\forall v \in E, \sigma_1(v) = \sigma_2(v)$

We denote this equality as $(\sigma_1, \tau_1) =_E (\sigma_2, \tau_2)$.

Lemma 5.1. *$\cdot =_E \cdot$ is an equivalence relation on configurations.*

We use this equality over configurations to define our equivalence relation for programs. Also see Figure 5.1 for a visual representation of the program equivalence.

$$\begin{tikzcd}
P_1 & P_2 \\
(\sigma_1, \tau) \arrow[r, leftrightarrow, "=_E"] \arrow[d, squiggly, "\tau'"'] & (\sigma_2, \tau) \arrow[d, squiggly, "\tau'"] \\
(\sigma'_1, \tau + \tau') \arrow[r, leftrightarrow, "=_E"] & (\sigma'_2, \tau + \tau')
\end{tikzcd}$$

Figure 5.1: Illustration of program equivalence

Before defining program equivalence formally, we need to introduce some intermediate definitions that will help us capture the behavior of the programs more precisely.

Definition 5.2 (Used and defined variables)**.** Let P be a SIP program.

We call *defined variables* in the usual way as the variables in the set $\mathbf{Defs}(P) \subseteq X$ defined inductively as follows:

$$\begin{aligned} \mathbf{Defs}(\texttt{out}(x)) &= \mathbf{Defs}(\texttt{skip}) = \emptyset \\ \mathbf{Defs}(x \texttt{ <- } \texttt{in}()) &= \mathbf{Defs}(x \texttt{ <- } e) = \{x\} \\ \mathbf{Defs}(\texttt{if } c \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end}) &= \mathbf{Defs}(P_1) \cap \mathbf{Defs}(P_2) \\ \mathbf{Defs}(P_1 \texttt{ ; } P_2) &= \mathbf{Defs}(P_1) \cup \mathbf{Defs}(P_2) \end{aligned}$$

Conversely, we call *used variables* the variables in the set $\mathbf{Use}(P) \subseteq X$ defined inductively as follows, where for any expression e, $\mathbf{Use}(e) \subseteq X$ denotes the variables appearing in e:

$$\begin{aligned} \mathbf{Use}(\texttt{out}(x)) &= \{x\} \\ \mathbf{Use}(x \texttt{ <- } \texttt{in}()) &= \mathbf{Use}(\texttt{skip}) = \emptyset \\ \mathbf{Use}(x \texttt{ <- } e) &= \mathbf{Use}(e) \\ \mathbf{Use}(\texttt{if } c \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end}) &= \mathbf{Use}(c) \cup \mathbf{Use}(P_1) \cup \mathbf{Use}(P_2) \\ \mathbf{Use}(P_1 \texttt{ ; } P_2) &= \mathbf{Use}(P_1) \cup \mathbf{Use}(P_2) \setminus \mathbf{Defs}(P_1) \end{aligned}$$

For a given program P, defined variables capture the set of variables which are *for sure* changed by the program. It is of importance to note that some other variables *may* be affected during the execution of the program, this is a consequence of the inductive definition in Definition 5.2 and especially the rule about `if` statements.

On the other hand, used variables capture the set of variables that are read before being assigned by the program. Note that following the definition of $\mathbf{Use}(P_1 \texttt{ ; } P_2)$, and the previous remark about defined variables, used variables are accessed, but they *may not* be assigned.

Definition 5.3 (Program equivalence)**.** Given two SIP programs P_1 and P_2, and given $E \subseteq X \cup Y$, we say that P_1 and P_2 are equivalent with respect to E (denoted $P_1 \sim_E P_2$) whenever:

1. $\mathbf{Use}(P_1) \cup \mathbf{Use}(P_2) \subseteq E$
2. $\mathbf{Rand}(P_1) = \mathbf{Rand}(P_2)$
3. $\forall \mathcal{A}.\ \forall c_1, c_2 \in \mathbf{State} \times \mathbf{Trace}.\ c_1 =_E c_2 \implies [\![P_1]\!]_{\mathcal{A}}(c_1) =_E [\![P_2]\!]_{\mathcal{A}}(c_2)$

Lemma 5.2. $\cdot \sim_E \cdot$ *is an equivalence relation.*

Proof. This is a consequence of Lemmas 4.1 and 5.1 and the fact that $\mathbf{Use}(P) \subseteq E$. The proof was checked using the theorem prover Lean4 [MU21]: see lemmas `Program.equivalent.trans`, `Program.equivalent.refl`, and `Program.equivalent.symm`. □

Note that the equivalence relation defined in Definition 5.3 is actually a congruence for programs. That is, let us consider arbitrary *contexts* (defined in Figure 5.2) as programs with *holes*. Given such a context C and a SIP program P, we denote by $C[P]$ the program resulting in the replacement of holes in C by copies of P then we get the congruence result exposed in Lemma 5.3.

Context	C	$::=$	x `<-` e	*assignment*
		$\mid$	x `<-` `in()`	*input*
		$\mid$	`out(`x`)`	*output*
		$\mid$	`skip`	*no operation*
		$\mid$	`if` e `then` C `else` C `end`	*conditional statement*
		$\mid$	C `;` C	*sequence*
		$\mid$	$\sqcup$	*hole*

Figure 5.2: Grammar for SIP contexts

Lemma 5.3. *Let C be a context as defined in Figure 5.2 such that* $\mathbf{Use}(C) \subseteq E$ *and let P_1, P_2 and E such that $P_1 \sim_E P_2$, we have:*

$$C[P_1] \sim_E C[P_2]$$

Proof. This is done by induction of the structure of C. This proof was checked using the theorem prover Lean4 [MU21]: : see lemmas `Program.equivalent.congr`. $\square$

Now we prove that two equivalent programs are *indistinguishable* from the attack point of view, that is, both the quantitative reachability and the leakage is preserved.

Lemma 5.4. *Let P_1 and P_2 be two programs such that $P_1 \sim_E P_2$ for some E. Then for any attack $\mathcal{A}$ it holds:*

1. $\mathcal{L}_{P_1}(\mathcal{A}) = \mathcal{L}_{P_2}(\mathcal{A})$
2. $\mathcal{R}^e_{P_1}(\mathcal{A}) = \mathcal{R}^e_{P_2}(\mathcal{A})$ *for any boolean expression e such that* $\mathbf{Use}(e) \subseteq E$

Proof. Let $\sigma \in \mathbf{State}_0$, and let $(\sigma_1, \tau_1) = [\![P_1]\!]_{\mathcal{A}}(\sigma, [])$ and $(\sigma_2, \tau_2) = [\![P_2]\!]_{\mathcal{A}}(\sigma, [])$. Then $(\sigma_1, \tau_1) =_E (\sigma_2, \tau_2)$ is a consequence of $P_1 \sim_E P_2$ and implies $\tau_1 = \tau_2$ and $\sigma_1(e) = \sigma_2(e)$.

Furthermore, as $\mathbf{Rand}(P_1) = \mathbf{Rand}(P_2)$, the counting mentioned in Properties 4.2 and 4.3 is done with respect to the same set of initial states $\mathbf{State}_0$. $\square$

5.2 Linearisation

The linearisation process for SIP essentially amounts to eliminating `if` constructs in the source program while producing equivalent `ite` expressions. In order to do so, we define a simple type system for SIP program that is sufficient to ensure that the linearisation does not alter the program's observable behavior.

5.2.1 Behavioral Type System

The type system for SIP programs (presented in Figure 5.3) essentially guarantees that the branches in `if` statements are balanced with regard to the interactions.

We will see later that being well-typed is a sufficient condition for successful linearisation. This means that whenever a program is well-typed, we can ensure that the linearisation process terminates. This allows to have a static (and easy to check) criterion on programs to determine whether they are linearisable.

$$\frac{}{\vdash \texttt{out}(x) : [o]}$$

(a) Output

$$\frac{}{\vdash x \texttt{ <- } \texttt{in}() : [i]}$$

(b) Input

$$\frac{\vdash P_1 : L \qquad \vdash P_2 : L}{\vdash \texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end} : L}$$

(c) Conditional statement

$$\frac{\vdash P : L \qquad \vdash P' : L'}{\vdash P \texttt{ ; } P' : L + L'}$$

(d) Sequence

$$\frac{}{\vdash x \texttt{ <- } e : []}$$

(e) Assignment

Figure 5.3: Behavioral type system

Remark 5.2.1. The fact that programs that are not well-typed are not linearisable needs to be discussed.

Whenever a program is not well-typed, it must be that two branches in the program yield different shapes of traces. In this case, one my say that the attacker is able to differentiate them (simply based on the fact that the interaction is different).

In this case, we need to recall the goal of the SIP language: be an formal intermediate representation for programs. This means that imposing restrictions on the shape of the trace is fine: one can build as many SIP programs as there are traces with different types, and analyse each program separately.

In our type systems, the type of a program is a sequence of markers (i and o) that are used to describe the alternation between inputs and outputs along the execution of the program. Rules "Input" (Figure 5.3b) and "Output" (Figure 5.3a) describe the types for input and output statements, being respectively $[i]$ and $[o]$. As an assignment does not produce any event, its type is the empty list. The "Sequence" (Figure 5.3d) rule is intuitive: the type of two programs executed one after the other is the concatenation (denoted $+$) of their types. The "Conditional statement" (Figure 5.3c) rule is interesting: we want to ensure that whatever the branch is taken, the sequence of interaction stays the same, which means that the types of the two branches has to be the same.

Definition 5.4 (Well-typedness)**.** A SIP program P is said to be *well-typed* whenever there exists a L such that:

$$\vdash P : L$$

5.2.2 Transformations

Definition 5.5 (Linearization)**.** Let P be a well-typed SIP program, we denote by $\mathbf{Lin}(P)$ the program obtained by:

1. Applying the `if` condition extraction (Figure 5.4a) to P;
2. Applying for all `if` statement, innermost first, all the other transformations (Figures 5.4b to 5.4d), removing empty `if` statements when both their branches are empty.

Note that the linearisation procedure terminates: at each application of a transformation we reduce the number of statements inside the innermost `if` statement by at least 1, until we remove that very `if` statement, decreasing the number of `if` statements in the program. Overall, at each step of the procedure, we either decrease the number of `if` statements, or the sum of the *depth* (with regard to the number of parent `if` statement) of all statements in the program, and this in turns ensures termination.

Type preservation We begin our study of the linearisation process by a proof that the overall structure of the interactions between the program and the attack is preserved, that is, the linearisation process preserves the type of the program.

Lemma 5.5. *Given a* SIP *program P such that $\vdash P : L$, let T be one of the transformations defined in Figures 5.4a to 5.4d, we have:*

$$\vdash T(P) : L$$

Proof. The cases of transformations in Figures 5.4a and 5.4c are trivial: the transformation only affects statements over which the type does not have any influence.

Let us first consider the transformation described in Figure 5.4d. By definition of the behavioral type system (see Figure 5.3), we have $\vdash P : [i] :: L$ with $\vdash P_1 : L$ and $\vdash P_2 : L$. This is exactly the type of the program after transformation.

The same reasoning applies to the transformation in Figure 5.4b. □

Corollary 5.6 (Type preservation)**.** *Given a* SIP *program P such that $\vdash P : L$, we have:*

$$\vdash \mathbf{Lin}(P) : L$$

Equivalence after linearisation We have proven that the linearisation process preserves the type of the program. We will now prove that actually, these operations transform the program into an equivalent one (as in Definition 5.3).

Lemma 5.7. *Given a well-typed* SIP *program P, let T be one of the transformations defined in Figures 5.4a to 5.4d. We have:*

$$P \sim_{\mathbf{Vars}(P)} T(P)$$

Proof. First note that for every transformation T we have:

$$\mathbf{Use}(T(P)) = \mathbf{Use}(P)$$
$$\mathbf{Rand}(T(P)) = \mathbf{Rand}(P)$$

It follows that Items 1 and 2 in Definition 5.3 hold. Thus it is only left to prove Item 3.

The proof will be carried using the relational semantics (i.e. Figure 4.3) for program, showing for each possible transformation that the transformed program and the original program are equivalent. Note that while this reasoning does directly use the relational semantics of SIP programs instead of the denotational semantics of the program, the reasoning still applies.

Furthermore, note that thanks to Lemmas 5.2 and 5.3, it is enough to consider the subprograms that are transformed in isolation. This is because the equivalence result on the whole program will be lifted from the equivalence of the subprogram.

Let us first consider Figure 5.4a, in this case the transformation only amounts to storing the value of the conditional in the `if` statement in a fresh variable, i.e. not belonging to $\mathbf{Vars}(P)$. In this case the property trivially holds after stepping into any of the two branches of the `if` statement.

We reason by cases on all the remaining transformations, and further on depending on the value of the condition of the transformed `if` statement. Let $\sigma_1, \sigma_2 \in \textbf{State}$ such that $\sigma_1\lfloor_{\mathbf{Vars}(P)} = \sigma_2\lfloor_{\mathbf{Vars}(P)}$ and $\tau \in \textbf{Trace}$.

Case "Assign translate" (Figure 5.4c): We only do the case where the assignment statement is in the left branch of the if statement, the other case following the same reasoning.

1. if $\sigma_1(p) = \sigma_2(p) = \top$, then we have:

$$(\texttt{if}\ p\ \texttt{then}\ x\ \texttt{<-}\ e\ \texttt{;}\ P_1\ \texttt{else}\ P_2\ \texttt{end}, \sigma_1, \tau) \underset{\mathcal{A}}{\longrightarrow} * \ (P_1, \sigma_1[x \leftarrow \sigma_1(e)], \tau)$$

   We can then compute the transitions for the transformed program:

$$\begin{aligned} &(x\ \texttt{<-}\ \texttt{ite}(p,\ e,\ x)\ \texttt{;}\ \texttt{if}\ p\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}, \sigma_2, \tau) \\ \underset{\mathcal{A}}{\longrightarrow}&(\texttt{if}\ p\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}, \sigma_2[x \leftarrow \sigma_2(e)], \tau) \\ \underset{\mathcal{A}}{\longrightarrow}&(P_1, \sigma_2[x \leftarrow \sigma_2(e)], \tau) \end{aligned}$$

2. Otherwise we have:

$$(\texttt{if}\ p\ \texttt{then}\ x\ \texttt{<-}\ e\ \texttt{;}\ P_1\ \texttt{else}\ P_2\ \texttt{end}, \sigma_1, \tau) \underset{\mathcal{A}}{\longrightarrow} * \ (P_2, \sigma_1, \tau)$$

   And the transitions of the transformed program are:

$$\begin{aligned} &(x\ \texttt{<-}\ \texttt{ite}(p,\ e,\ x)\ \texttt{;}\ \texttt{if}\ p\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}, \sigma_2, \tau) \\ \underset{\mathcal{A}}{\longrightarrow}&(\texttt{if}\ p\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}, \sigma_2, \tau) \\ \underset{\mathcal{A}}{\longrightarrow}&(P_2, \sigma_2, \tau) \end{aligned}$$

In both cases the resulting states are obviously equal on the variables in $\mathbf{Vars}(P)$.

Case "In translate" (Figure 5.4d):

1. if $\sigma_1(p) = \sigma_2(p) = \top$, we have for the original program:

$$\begin{aligned} &(\texttt{if}\ p\ \texttt{then}\ x\ \texttt{<-}\ \texttt{in()}\ \texttt{;}\ P_1\ \texttt{else}\ y\ \texttt{<-}\ \texttt{in()}\ \texttt{;}\ P_2\ \texttt{end}, \sigma_1, \tau) \\ \underset{\mathcal{A}}{\longrightarrow}&(x\ \texttt{<-}\ \texttt{in()}\ \texttt{;}\ P_1, \sigma_1, \tau) \\ \underset{\mathcal{A}}{\longrightarrow}&(P_1, \sigma_1[x \leftarrow \mathcal{A}(\tau)], \tau :: \mathrm{in}(\mathcal{A}(\tau))) \end{aligned}$$

Then, for the transformed program (with $S = \texttt{if}\ p\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}$):

$$\begin{aligned}
&(p' \texttt{ <- in() ; } x \texttt{ <- ite(}p, p', x\texttt{) ; } y \texttt{ <- ite(}p, y, p'\texttt{) ; } S, \sigma_2, \tau)\\
\underset{\mathcal{A}}{\longrightarrow}&(x \texttt{ <- ite(}p, p', x\texttt{) ; } y \texttt{ <- ite(}p, y, p'\texttt{) ; } S, \sigma_2\big[p' \leftarrow \mathcal{A}(\tau)\big], \tau :: \mathrm{in}(\mathcal{A}(\tau)))\\
\underset{\mathcal{A}}{\longrightarrow}&(y \texttt{ <- ite(}p, y, p'\texttt{) ; } S, \sigma_2\big[p' \leftarrow \mathcal{A}(\tau)\big][x \leftarrow \mathcal{A}(\tau)], \tau :: \mathrm{in}(\mathcal{A}(\tau)))\\
\underset{\mathcal{A}}{\longrightarrow}&(\texttt{if}\ p\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}, \sigma_2\big[p' \leftarrow \mathcal{A}(\tau)\big][x \leftarrow \mathcal{A}(\tau)], \tau :: \mathrm{in}(\mathcal{A}(\tau)))\\
\underset{\mathcal{A}}{\longrightarrow}&(P_1, \sigma_2\big[p' \leftarrow \mathcal{A}(\tau)\big][x \leftarrow \mathcal{A}(\tau)], \tau :: \mathrm{in}(\mathcal{A}(\tau)))
\end{aligned}$$

2. otherwise if $\sigma_1(p) = \sigma_2(p) = \bot$ the same reasoning applies.

In both cases, at the end of the execution, we indeed find the expected property that:

$$\sigma_1[x \leftarrow \mathcal{A}(\tau)] =_{\mathbf{Vars}(P)} \sigma_2\big[p' \leftarrow \mathcal{A}(\tau)\big][x \leftarrow \mathcal{A}(\tau)]$$

Case "Out translate" (Figure 5.4b):

1. if $\sigma_1(p) = \sigma_2(p) = \top$, we have for the original program:

$$\begin{aligned}
&(\texttt{if}\ p\ \texttt{then out(}x\texttt{) ; } P_1\ \texttt{else out(}y\texttt{) ; } P_2\ \texttt{end}, \sigma_1, \tau)\\
\underset{\mathcal{A}}{\longrightarrow}&(\texttt{out(}x\texttt{) ; } P_1, \sigma_1, \tau)\\
\underset{\mathcal{A}}{\longrightarrow}&(P_1, \sigma_1, \tau :: \mathrm{out}(\sigma_1(x)))
\end{aligned}$$

Then, for the transformed program (with $S = \texttt{if}\ p\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}$):

$$\begin{aligned}
&(p' \texttt{ <- ite(}p, x, y\texttt{) ; out(}p'\texttt{) ; } S, \sigma_2, \tau)\\
\underset{\mathcal{A}}{\longrightarrow}&(\texttt{out(}p'\texttt{) ; } S, \sigma_2\big[p' \leftarrow \sigma_2(x)\big], \tau)\\
\underset{\mathcal{A}}{\longrightarrow}&(\texttt{if}\ p\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}, \sigma_2\big[p' \leftarrow \sigma_2(x)\big], \tau :: \mathrm{out}(\sigma_2(x)))\\
\underset{\mathcal{A}}{\longrightarrow}&(P_1, \sigma_2\big[p' \leftarrow \sigma_2(x)\big], \tau :: \mathrm{out}(\sigma_2(x)))
\end{aligned}$$

2. otherwise if $\sigma_1(p) = \sigma_2(p) = \bot$ the same reasoning applies.

Similarly to the previous cases, the two states are equal on the variables in **Vars**(P). Furthermore, as the states are equal on **Vars**(P), they are we have in particular $\sigma_1(x) = \sigma_2(x)$ and the traces are equal.

□

Corollary 5.8. *Given a well-typed* SIP *program P, we have:*

$$P \sim_{\mathbf{Vars}(P)} \mathbf{Lin}(P)$$

Lemma 5.9. *Given P a well-typed* SIP *program,* **Lin**(P) *does not contain any* ***if*** *statement.*

Proof. This is by induction on the number of `if` statements in P. The base case is trivial, let us thus consider the case where P contains $n+1$ `if` statements.

Take the innermost `if` statement, by definition of well-typedness (see Figure 5.3) the two branches of the `if` statement follow the same alternation of `in` and `out` statements.

One can thus repeatedly apply the transformation in Figure 5.4c until the first `in` (resp. `out`) statement appears as the first statement of the left branch, and repeat the same process for the right branch until the corresponding `in` (resp. `out`) appears as its first statement. Apply then the transformation in Figure 5.4d (resp. Figure 5.4b), removing the first statement in both branches. This new *partially* linearised `if` statement is still well-typed, and the process can be carried until both branches are empty, and the statement can be removed.

The innermost `if` statement being removed, the program now has n `if` statements, and the induction hypothesis applies. □

Theorem 5.10 (Correctness of linearisation). *Given P a well typed* SIP *program, then* $\mathbf{Lin}(P)$ *is an equivalent program on* $\mathbf{Vars}(P)$ *with the same type and no* `if` *statements.*

We are now able to state the theorems of interest in this section. They informally mean that one can study the linearised program in order to synthesize an attack on the original program. These two theorems are a direct consequence of Lemmas 5.4 and 5.7.

Theorem 5.11. *Given a well-typed program P and a boolean expression e over the variables of P, we have:*

$$\forall \mathcal{A}.\ \mathcal{R}^e_P(\mathcal{A}) = \mathcal{R}^e_{\mathbf{Lin}(P)}(\mathcal{A})$$

Theorem 5.12. *Given a well-typed program P we have:*

$$\forall \mathcal{A}.\ \mathcal{L}_P(\mathcal{A}) = \mathcal{L}_{\mathbf{Lin}(P)}(\mathcal{A})$$

5.3 Single static assignment

Single static assignment (SSA) is a special form for programs such that each variable of the program is assigned exactly once. This is standard intermediate representation in compilers and formal verifiers [ASU86], and it greatly simplifies the analysis of the program's behavior.

Transformation to SSA form is a standard operation in computer science, and it is well-known that this transformation preserves the values of the variables of the program.

In particular for a linearised program, it is possible to define an SSA transformation such that the following holds:

$$P \sim_{\mathbf{Vars}(P)} SSA(P)$$

The idea is to compute the SSA form starting from the last statement of the program. This way the values of the variables at the very end of the program will be set to their correct values at the end of the execution.

$$\frac{\texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end}}{\begin{array}{l} p \texttt{ <- } e \\ \texttt{if } p \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end} \end{array}}$$

(a) Split if

$$\frac{\texttt{if } p \texttt{ then out}(x) \texttt{ ; } P_1 \texttt{ else out}(y) \texttt{ ; } P_2 \texttt{ end}}{\begin{array}{l} p' \texttt{ <- ite}(p\texttt{, } x\texttt{, } y) \\ \texttt{out}(p') \\ \texttt{if } p \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end} \end{array}}$$

(b) Out translate

$$\frac{\texttt{if } p \texttt{ then } x \texttt{ <- } e \texttt{ ; } P_1 \texttt{ else } P_2 \texttt{ end}}{\begin{array}{l} x \texttt{ <- ite}(p\texttt{, } e\texttt{, } x) \\ \texttt{if } p \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end} \end{array}} \qquad \frac{\texttt{if } p \texttt{ then } P_1 \texttt{ else } x \texttt{ <- } e \texttt{ ; } P_2 \texttt{ end}}{\begin{array}{l} x \texttt{ <- ite}(p\texttt{, } x\texttt{, } e) \\ \texttt{if } p \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end} \end{array}}$$

(c) Assign translate

$$\frac{\texttt{if } p \texttt{ then } x \texttt{ <- in() ; } P_1 \texttt{ else } y \texttt{ <- in() ; } P_2 \texttt{ end}}{\begin{array}{l} p' \texttt{ <- in()} \\ x \texttt{ <- ite}(p\texttt{, } p'\texttt{, } x) \\ y \texttt{ <- ite}(p\texttt{, } y\texttt{, } p') \\ \texttt{if } p \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end} \end{array}}$$

(d) In translate

Figure 5.4: Program linearisation transformations

Note that traditionally, SSA requires that all variables are assigned before being used. In our case we relax this requirement to all variables except those in **Rand**(P). We thus assume without loss of generality that variables in **Use**(P) (if any) are initially explicitly assigned the value $0_{\mathbf{Val}}$.

5.4 Bit-blasting

In our work, we restrict to source programs where **Val** is the set of fix-sized bit-vectors. This theory is well-studied and many program analysis tools[CDE08; Dav+16; Sho+16] use it as a back-end to represent program states and reason about them.

The process of bit-blasting consists of transforming a program acting on bit-vectors to boolean values. This is possible because one can simply encode all the operations on every bit and explicitly encode the circuits computing the various operations directly within the program.

This comes at a cost: the size of the program may increase, and many temporary variables will be added.

Example 5.4.1. Let the following program P be on bit-vectors of size 2:

```
x1 <- in()
x2 <- x1 + y
out(x2)
```

Then the bit-blasted version of the program is (where $\oplus$ denote boolean exclusive or):

```
x1_1 <- in()
x1_2 <- in()

x2_1 <- x1_1 ⊕ y_1
z1 <- x1_1 ∧ y_1
z2 <- x1_2 ⊕ y_2
x2_2 <- z2 ⊕ z1

out(x2_1)
out(x2_2)
```

Note that this transformation is different that the other. In this case, we change the domain of values **Val**: from bit-vectors to booleans. However, it is easy to see that one can convert any attack from (resp. to) booleans to (resp. from) bit-vectors: this is just a matter of rewriting. Thus one can see that it is sufficient to consider the bit-blasted program, synthesize an attack on that program, then translate the resulting attack back to bit-vectors.

5.5 Normal form and characteristic formula of a program

First, we define a symbolic version of the trace of a program. This is done inductively on the program and will help when defining normal forms.

Definition 5.6 (Symbolic trace)**.** Given a SIP program P, we define the *symbolic trace* of P inductively as follows:

$$\begin{aligned}
\mathrm{tr}(x \texttt{ <- } \texttt{in}\texttt{()}) &\triangleq [\mathrm{in}(x)] \\
\mathrm{tr}(\texttt{out}\texttt{(}x\texttt{)}) &\triangleq [\mathrm{out}(x)] \\
\mathrm{tr}(x \texttt{ <- } e) &\triangleq [] \\
\mathrm{tr}(P_1 \texttt{ ; } P_2) &\triangleq \mathrm{tr}(P_1) + \mathrm{tr}(P_2)
\end{aligned}$$

Given a state $\sigma \in \mathbf{State}$, we define $\sigma(\mathrm{tr}(P))$ as the concrete trace obtained by evaluating the variables in the symbolic trace to their values in σ.

We can now define what SIP programs *normal forms* are. We will then discuss why they are of importance in our study.

Definition 5.7 (Normal SIP program)**.** A SIP program P is said to be in *normal form* if:

1. P is *linear* (i.e. it does not contain any `if` statement)
2. P is in SSA form
3. P acts over boolean variables only
4. The variables appearing in $\mathrm{tr}(P)$ are $(x_i)_{i\in\{1,\ldots,n\}}$ and:
   - Any variable appears exactly once
   - They appear in the order $x_1, \ldots, x_n$ within the symbolic trace

Remark 5.5.1. When a program P is in normal form, for any variable $x_i \in \mathbf{In}(P) \cup \mathbf{Out}(P)$ the index i corresponds to its position in the trace.

Theorem 5.13. *For any well-typed* SIP *program over bit-vectors P, there exists an equivalent* SIP *program* $\mathbf{Norm}(P)$ *in normal form.*

Proof. Let us take each item in Definition 5.7, and prove that one can construct a program step by step while keeping it equivalent to the original program. The resulting program will therefore be in normal form.

Item 1 is a consequence of Theorem 5.10.

Item 2 is discussed in Section 5.3. Note that transforming to SSA form preserves Item 1.

Item 3 is discussed in Section 5.4, note though that now the program and the attacker are both over the booleans. It is clear that bit-blasting can be done such that Items 1 and 2 are preserved.

Assume that the program satisfies Items 1 to 3, we prove that we can construct an equivalent program satisfying Item 4 while preserving the other items. We can add any number of new variables that follow the trace numbering. If the variable is supposed to be in $\mathbf{In}(P)$, we input from the attack using our new variable and then assign it to the old variable. The case of $\mathbf{Out}(P)$ is similar: we assign the value of the old variable to the new variable, and we then output the new variable. The update program is obviously equivalent to the original one and that concludes the proof. □

Example 5.5.2. The following program P is in normal form:

```
x1 <- y1 ∨ y2
out(x1)
x2 <- in()
x3 <- x2 ∧ y2
out(x3)
x4 <- in()
```

Furthermore, its symbolic trace is:

$$[\mathrm{out}(x_1), \mathrm{in}(x_2), \mathrm{out}(x_3), \mathrm{in}(x_4)]$$

We now define the characteristic formula of a program in normal form. Intuitively, the formula captures the state space *after* execution of the program, leaving the attack undetermined. This is a key step towards the encoding of the security problems of interest in this thesis towards synthesizing attack. This point will be brought further later in the thesis, as the encoding of a security problem depends on the capacities (in terms of knowledge gathering) of the attack to be synthesized.

Definition 5.8. Given a SIP program P in normal form, we define the *characteristic formula* of P, denoted $\phi(P)$, inductively as follows:

$$\begin{aligned}\phi(x_k \texttt{ <- } \texttt{in}()) &\triangleq \top\\ \phi(\texttt{out}(x_k)) &\triangleq \top\\ \phi(x \texttt{ <- } e) &\triangleq x \Leftrightarrow e\\ \phi(P_1 \texttt{ ; } P_2) &\triangleq \phi(P_1) \wedge \phi(P_2)\end{aligned}$$

Example 5.5.2 (continuing from p. 60). The characteristic formula of P is:

$$(x_1 \Leftrightarrow y_1 \vee y_2) \wedge (x_3 \Leftrightarrow x_2 \wedge y_2)$$

We can now state a key theorem about the relationship between the characteristic formula of a program and the set of its final states. It is a key step in the proofs of correctness of the various reductions to counting problems studied throughout this thesis.

Theorem 5.14. *Given a* SIP *program P in normal form. The following are equivalent for every attack $\mathcal{A}$ and configuration (σ, τ):*

1. $(\sigma, \tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0)$
2. $\sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}(\tau_{<i})$ *and* $\tau = \sigma(\mathrm{tr}(P))$

Proof. We prove this by induction on the length of the program.

Base case: programs P of length 1 Because of the restrictions imposed on programs in normal form, we have only two sub-cases. This is because any `out(`x_k`)` statement has to be such that $x_k \notin \mathbf{Rand}(P)$, and furthermore, x_k has to be initialized before being used (per SSA requirements). This means that any program containing an output statement has to be of length at least 2 (one assignment and the output statement).

Case $P = x$ `<-` e:

$$\begin{aligned}(\sigma, \tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) &\iff (\sigma, \tau) \in [\![x \texttt{ <- } e]\!]_{\mathcal{A}}(\mathbf{State}_0)\\ &\iff \sigma(x) = \sigma(e) \text{ and } \tau = []\\ &\iff \sigma \models x = e \text{ and } \tau = \sigma([])\\ &\iff \sigma \models \phi(P) \wedge \bigwedge\nolimits_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}(\tau_{<i}) \text{ and } \tau = \sigma(\mathrm{tr}(P))\end{aligned}$$

Case $P = x_1$ `<- in()`:

$$\begin{aligned}(\sigma, \tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) &\iff (\sigma, \tau) \in [\![x_1 \texttt{ <- in()}]\!]_{\mathcal{A}}(\mathbf{State}_0)\\ &\iff \sigma(x_1) = \mathcal{A}([]) \text{ and } \tau = [\mathrm{in}(\mathcal{A}([]))]\\ &\iff \sigma \models x_1 = \mathcal{A}(\tau_{<1}) \text{ and } \tau = [\mathrm{in}(\sigma(x_1))]\\ &\iff \sigma \models \phi(P) \wedge \bigwedge\nolimits_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}(\tau_{<i}) \text{ and } \tau = \sigma(\mathrm{tr}(P))\end{aligned}$$

Induction step: $P = P'$; S with P' of length m We assume that the property is true for all programs of length smaller or equal to m. This means that the following are equivalent (this is the induction hypothesis for program P')

- $(\sigma, \tau) \in [\![P']\!]_{\mathcal{A}}(\mathbf{State}_0)$
- $\sigma \models \phi(P') \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau_{<i})$ and $\tau = \sigma(\mathrm{tr}(P'))$

Furthermore, note that the following is true with regard to P:

$$\begin{aligned}\mathbf{In}(P) &= \mathbf{In}\big(P'\big) \cup \mathbf{In}(S)\\ \mathrm{tr}(P) &= \mathrm{tr}\big(P'\big) + \mathrm{tr}(S)\end{aligned}$$

We then reason by cases on S.

Case $S = \texttt{out}(x_k)$: In this case recall that $\mathbf{In}(P) = \mathbf{In}(P')$ and $\mathrm{tr}(P) = \mathrm{tr}(P') + [\mathrm{out}(x_k)]$:

$$
\begin{aligned}
(\sigma,\tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) &\iff (\sigma,\tau) \in [\![P' \ \texttt{;}\ S]\!]_{\mathcal{A}}(\mathbf{State}_0) \\
&\iff (\sigma,\tau) \in [\![S]\!]_{\mathcal{A}}\big([\![P']\!]_{\mathcal{A}}(\mathbf{State}_0)\big) \\
&\iff \left|\begin{array}{l} (\sigma,\tau) = [\![S]\!]_{\mathcal{A}}(\sigma',\tau') \\ (\sigma',\tau') \in [\![P']\!]_{\mathcal{A}}(\mathbf{State}_0) \end{array}\right. \\
&\iff \left|\begin{array}{ll} (\sigma,\tau) = [\![S]\!]_{\mathcal{A}}(\sigma',\tau') & \sigma' \models \phi(P') \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau'_{<i}) \\ & \tau' = \sigma'(\mathrm{tr}(P')) \end{array}\right. \\
&\iff \left|\begin{array}{ll} (\sigma,\tau) = [\![\texttt{out}(x_k)]\!]_{\mathcal{A}}(\sigma',\tau') & \sigma' \models \phi(P') \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau'_{<i}) \\ & \tau' = \sigma'(\mathrm{tr}(P')) \end{array}\right. \\
&\iff \left|\begin{array}{ll} \sigma = \sigma' & \sigma' \models \phi(P') \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau'_{<i}) \\ \tau = \tau' + [\mathrm{out}(\sigma'(x_k))] & \tau' = \sigma'(\mathrm{tr}(P')) \end{array}\right. \\
&\iff \left|\begin{array}{l} \sigma \models \phi(P') \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau_{<i}) \\ \tau = \sigma(\mathrm{tr}(P')) + [\mathrm{out}(\sigma(x_k))] \end{array}\right. \\
&\iff \left|\begin{array}{l} \sigma \models \phi(P' \ \texttt{;}\ \texttt{out}(x_k)) \wedge \bigwedge_{x_i \in \mathbf{In}(P' \ \texttt{;}\ \texttt{out}(x_k))} x_i = \mathcal{A}(\tau_{<i}) \\ \tau = \sigma(\mathrm{tr}(P' \ \texttt{;}\ \texttt{out}(x_k))) \end{array}\right. \\
&\iff \left|\begin{array}{l} \sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}(\tau_{<i}) \\ \tau = \sigma(\mathrm{tr}(P)) \end{array}\right.
\end{aligned}
$$

Case $S = x \ \texttt{<-}\ e$: In this case $\mathbf{In}(P) = \mathbf{In}(P')$ and $\mathrm{tr}(P) = \mathrm{tr}(P')$:

$$
\begin{aligned}
(\sigma,\tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) &\iff \dots \\
&\iff \left|\begin{array}{ll} (\sigma,\tau) = [\![x \ \texttt{<-}\ e]\!]_{\mathcal{A}}(\sigma',\tau') & \sigma' \models \phi(P') \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau'_{<i}) \\ & \tau' = \sigma'(\mathrm{tr}(P')) \end{array}\right. \\
&\iff \left|\begin{array}{ll} \sigma = \sigma'[x \leftarrow \sigma'(e)] & \sigma' \models \phi(P') \wedge x = e \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau_{<i}) \\ \tau = \tau' & \tau' = \sigma'(\mathrm{tr}(P')) \end{array}\right. \\
&\iff \left|\begin{array}{l} \sigma \models \phi(P') \wedge x = e \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau_{<i}) \\ \tau = \sigma(\mathrm{tr}(P')) \end{array}\right. \\
&\iff \left|\begin{array}{l} \sigma \models \phi(P' \ \texttt{;}\ x \ \texttt{<-}\ e) \wedge \bigwedge_{x_i \in \mathbf{In}(P' \ \texttt{;}\ x \ \texttt{<-}\ e)} x_i = \mathcal{A}(\tau_{<i}) \\ \tau = \sigma(\mathrm{tr}(P' \ \texttt{;}\ x \ \texttt{<-}\ e)) \end{array}\right. \\
&\iff \left|\begin{array}{l} \sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}(\tau_{<i}) \\ \tau = \sigma(\mathrm{tr}(P)) \end{array}\right.
\end{aligned}
$$

Case $S = x_k$ `<- in()`: In this case $\mathbf{In}(P) = \mathbf{In}(P') \cup \{x_k\}$ and $\mathrm{tr}(P) = \mathrm{tr}(P') + [\mathrm{in}(x_k)]$:

$$(\sigma,\tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) \iff \dots$$

$$\iff \left| \begin{array}{ll} (\sigma,\tau) = [\![x_k \texttt{ <- in()}]\!]_{\mathcal{A}}(\sigma',\tau') & \sigma' \models \phi(P') \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau'_{<i}) \\ & \tau' = \sigma'(\mathrm{tr}(P')) \end{array} \right.$$

$$\iff \left| \begin{array}{ll} \sigma = \sigma'[x_k \leftarrow \mathcal{A}(\tau')] & \sigma' \models \phi(P') \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau'_{<i}) \\ \tau = \tau' + [\mathrm{in}(\mathcal{A}(\tau'))] & \tau' = \sigma'(\mathrm{tr}(P')) \end{array} \right.$$

$$\iff \left| \begin{array}{l} \sigma \models \phi(P') \wedge \bigwedge_{x_i \in \mathbf{In}(P')} x_i = \mathcal{A}(\tau_{<i}) \wedge x_k = \mathcal{A}(\tau_{<k}) \\ \tau = \sigma(\mathrm{tr}(P')) + [\mathrm{in}(\sigma(x_k))] \end{array} \right.$$

$$\iff \left| \begin{array}{l} \sigma \models \phi(P' \texttt{ ; } x_k \texttt{ <- in()}) \wedge \bigwedge_{x_i \in \mathbf{In}(P' \texttt{ ; } x_k \texttt{ <- in()})} x_i = \mathcal{A}(\tau_{<i}) \\ \tau = \sigma(\mathrm{tr}(P' \texttt{ ; } x_k \texttt{ <- in()})) \end{array} \right.$$

$$\iff \left| \begin{array}{l} \sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}(\tau_{<i}) \\ \tau = \sigma(\mathrm{tr}(P)) \end{array} \right.$$

□

Part III

Non-adaptive attacks

Chapter 6

Reducing to Max#SAT

We begin our study of attack synthesis by non-adaptive attacks. Non-adaptive attacks are informally those where the information received from the target program and the attack are not used. They thus only keep track of *which* interaction they are currently doing. The objective of this chapter is reducing non-adaptive attack synthesis to Max#SAT. This will in turn allow focusing our study on Max#SAT itself and resolution methods.

Definition 6.1 (Non-adaptive attacks)**.** An attack $\mathcal{A}$ is *non-adaptive* whenever:

$$\forall \tau_1, \tau_2.\ \|\tau_1{\downharpoonleft_{\text{in}}}\| = \|\tau_2{\downharpoonleft_{\text{in}}}\| \Rightarrow \mathcal{A}(\tau_1) = \mathcal{A}(\tau_2)$$

Therefore, any non-adaptive attack is fully characterized by a sequence of values $(a_1, \dots, a_n)$. Because we study programs in normal form, we can assume without loss of generality that the sequence of values (a_i) is labeled such that:

$$\mathcal{A}(\tau_{<i}) = a_i$$

Example 6.0.1. Any attack of the form $\tau \mapsto c$ for $c \in \mathbf{Val}$ is a non-adaptive attack. The attack defined by $\mathcal{A} \triangleq \tau \mapsto \|\tau{\downharpoonleft_{\text{in}}}\|$ is a non-adaptive attack over the integers.

The problem of attack synthesis (i.e. Definitions 4.13 and 4.15) thus amounts, in this case, to the synthesis of constants that maximize the target effectiveness. We prove in this chapter that every security criterion of interest in the thesis can in fact be reduced to instances of the so called Max#SAT problem [FRS17; Vig+22].

6.1 The Max#SAT problem

We recall here the definition of Max#SAT, together some preliminary definitions to be able to state the target reductions.

We assume three disjoint sets of variables X, Y, and Z respectively named *witness*, *counting* and *intermediate* variables. We often call *witnesses* the valuations over X.

Definition 6.2 (Induced set). Given a formula $\phi(X, Y, Z)$ and $x : X \to \mathbb{B}$ the set of models over Y *induced* by x is:

$$\mathcal{I}_{\phi}^{Y}(x) \triangleq \{y : Y \to \mathbb{B} \mid \exists z : Z \to \mathbb{B}.\ \phi(x, y, z)\}$$

We extend this definition to partial witnesses as follows, for some $E \subseteq X$:

$$\mathcal{I}_{\phi}^{Y}(x\lfloor_E) \triangleq \bigcup_{x' \in [x]_E} \mathcal{I}_{\phi}^{Y}(x')$$

Definition 6.3 (Count of a witness). Given a formula $\phi(X, Y, Z)$, the *count* of a witness x is defined by the cardinal of the set it induces, that is:

$$\|\phi(x)\|_Y \triangleq \left|\mathcal{I}_{\phi}^{Y}(x)\right|$$

This definition is extended naturally to partial witnesses, that is:

$$\|\phi(x\lfloor_E)\|_Y \triangleq \left|\bigcup_{x' \in [x]_E} \mathcal{I}_{\phi}^{Y}(x')\right|$$

Definition 6.4 (Max#SAT). Given a formula $\phi(X, Y, Z)$, the Max#SAT problem

$$\mathsf{M}X.\ \text{Я}Y.\ \exists Z.\ \phi$$

Asks for finding $x_m \in \mathcal{M}_X(\phi)$ such that:

$$\|\phi(x_m)\|_Y = \max_{x \in \mathcal{M}_X(\phi)} \|\phi(x)\|_Y$$

Example 6.1.1 (A Max#SAT instance). Let $X \triangleq \{a, b\}$, $Y \triangleq \{c\}$, $Z \triangleq \{d\}$, and $\phi \triangleq (\neg a \vee c) \wedge (a \vee \neg b) \wedge (\neg b \vee d)$, one can write the following Max#SAT problem:

$$\mathsf{M}a.\ \mathsf{M}b.\ \text{Я}c.\ \exists d.\ \phi(a, b, c, d)$$

We can then enumerate all the possible valuations over X and manually count their corresponding number of models over Y:

Model X	Resulting formula	Number of models over Y
$a \wedge b$	$c \wedge d$	1
$a \wedge \neg b$	c	1
$\neg a \wedge b$	$\bot$	0
$\neg a \wedge \neg b$	$\top$	2

We can thus clearly see that the optimal valuation over X is $\{a \mapsto \bot, b \mapsto \bot\}$.

6.2 Quantitative reachability

As mentioned previously in this chapter, the non-adaptive attack synthesis essentially amounts to the synthesis of constants to be given to the program at each `input` statement.

It follows from Property 4.2 that one can reduce the problem of non-adaptive attack synthesis to an instance of a MAX#SAT problem based on the characteristic formula of the program.

Theorem 6.1 (Quantitative reachability to Max#SAT). *Given a program P in normal form, and e a boolean expression. Let $(a_1^\star, \dots, a_n^\star) \in \mathbb{B}^n$. The following are equivalent:*

(1) The non-adaptive attack $\mathcal{A}^\star(\tau_{<i}) = a_i^\star$ is optimal with regard to the quantitative reachability objective e.

(2) The valuation $x_{\mathcal{A}^\star} = \{x_i \mapsto a_i^\star\}_{x_i \in \mathbf{In}(P)}$ is the answer to the MAX#SAT query

$$\mathsf{M}\mathbf{In}(P).\ \text{Я}\mathbf{Rand}(P).\ \exists\mathbf{Rest}(P).\ \exists\mathbf{Out}(P).\ \phi(P) \wedge e$$

Proof. Let $\mathcal{A}$ be a non-adaptive attack, i.e. such that $\mathcal{A}(\tau_{<i}) = a_i$ and let the associated valuation $x_{\mathcal{A}} = \{x_i \mapsto a_i\}_{x_i \in \mathbf{In}(P)}$. Note that following Property 4.2 it is sufficient to prove:

$$|\{(\sigma, \tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) \mid \sigma(e) = \top\}| = \|(\phi(P) \wedge e)(x_{\mathcal{A}})\|_{\mathbf{Rand}(P)} \tag{6.1}$$

Indeed, the quantitative reachability of $\mathcal{A}$ is proportional to $|\{(\sigma, \tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) \mid \sigma(e) = \top\}|$ and thus orders between assignments (i.e. witnesses in the MAX#SAT world) and attacks would be the same. This would ensure that the two optima correspond to one another.

Let us then prove Equation (6.1), the following is a consequence of Theorem 5.14:

$$\begin{aligned}
|\{(\sigma, \tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) \mid \sigma(e) = \top\}| &= \left|\left\{ (\sigma, \tau) \;\middle|\; \begin{array}{l} \tau = \sigma(\mathrm{tr}(P)) \\ \sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}(\tau_{<i}) \wedge e \end{array} \right\}\right| \\
&= \left|\left\{ \sigma \;\middle|\; \sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}(\tau_{<i}) \wedge e \right\}\right| \\
&= \left|\left\{ \sigma \;\middle|\; \sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = a_i \wedge e \right\}\right| \\
&= |\{\sigma \mid \sigma \models (\phi(P) \wedge e)[x_i \leftarrow a_i]\}| \\
&= \left|\left\{ \sigma\!\downharpoonright_{\mathbf{Rand}(P)} \;\middle|\; \sigma \models (\phi(P) \wedge e)[x_i \leftarrow a_i] \right\}\right| \\
&= \left| \mathcal{I}^{\mathbf{Rand}(P)}_{\phi(P) \wedge e}(x_{\mathcal{A}}) \right| \\
&= \|(\phi(P) \wedge e)(x_{\mathcal{A}})\|_{\mathbf{Rand}(P)}
\end{aligned}$$

□

Example 6.2.1. Let us consider the target expression $x_4 = y_1 \vee y_2$ and the following program:

```
x1 <- y1 ∨ y2
out(x1)
x2 <- in()
x3 <- x2 ∧ y2
out(x3)
x4 <- in()
```

The MAX#SAT problem corresponding to the quantitative reachability problem e is the following:

$$\mathsf{M} x_2, x_4.\ \mathsf{R} y_1, y_2.\ \exists x_1, x_3.\ (x_1 \Leftrightarrow y_1 \vee y_2) \wedge (x_3 \Leftrightarrow x_2 \wedge y_2) \wedge (x_4 \Leftrightarrow y_1 \vee y_2)$$

One can then use this query to synthesize the optimal non-adaptive attack with regard to the program mentioned above. The optimal valuation found would be $x^\star = \{x_2 \mapsto \bot, x_4 \mapsto \top\}$, with a count of 3. This in turns yields the following non-adaptive attack:

$$\mathcal{A}^\star(\tau) = \begin{cases} \bot & \text{if } \|\tau\lfloor_{\text{in}}\| = 1 \\ \top & \text{otherwise} \end{cases}$$

6.3 Leakage

Let us now consider the problem of synthesis of attack synthesis with regard to a leakage objective. In this case, because of Property 4.3 the actual computation of leakage is directly reduced to a counting problem. We will thus perform a reduction in the same fashion as in the previous section.

Theorem 6.2 (Leakage to Max#SAT). *Given a program P in normal form, let $(a_1^\star, \dots, a_n^\star) \in \mathbb{B}^n$. The following are equivalent:*

- *The non-adaptive attack $\mathcal{A}^\star(\tau_{<i}) = a_i^\star$ is optimal with regard to leakage.*
- *The valuation $x_{\mathcal{A}^\star} = \{x_i \mapsto a_i^\star\}_{x_i \in \mathbf{In}(P)}$ is the answer to the MAX#SAT query*

$$\mathsf{M}\mathbf{In}(P).\ \mathsf{R}\mathbf{Out}(P).\ \exists\mathbf{Rest}(P).\ \exists\mathbf{Rand}(P).\ \phi(P)$$

Proof. Let $\mathcal{A}$ be a non-adaptive attack, i.e. such that $\mathcal{A}(\tau_{<i}) = a_i$, and let $x_{\mathcal{A}} = \{x_i \mapsto a_i\}_{x_i \in \mathbf{In}(P)}$ be the corresponding witness.

Similarly to Theorem 6.1, it follows from Property 4.3 that it is sufficient to prove:

$$\left|[\![P]\!]_{\mathcal{A}}\lfloor_{\mathbf{Trace}}\lfloor_{\text{out}}(\mathbf{State}_0)\right| = \|\phi(P)(x_{\mathcal{A}})\|_{\mathbf{Out}(P)}$$

In this case this is because $\log_2$ is order preserving. Proving this will prove that the orders upon witnesses and attack correspond, and thus prove that the maxima also correspond.

The following is a consequence of Theorem 5.14:

$$
\begin{aligned}
\left|[\![P]\!]_{\mathcal{A}}\restriction_{\mathbf{Trace}}\restriction_{\text{out}}(\mathbf{State}_0)\right| &= |\{\tau\restriction_{\text{out}} \mid \exists\sigma.\ (\sigma,\tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0)\}| \\
&= \left|\left\{\tau\restriction_{\text{out}} \;\middle|\; \exists\sigma. \;\middle|\begin{array}{l}\tau = \sigma(\text{tr}(P)) \\ \sigma \models \phi(P) \wedge \bigwedge_{i=1,n} x_i = \mathcal{A}(\tau_{<i})\end{array}\right\}\right| \\
&= \left|\left\{\sigma(\text{tr}(P))\restriction_{\text{out}} \;\middle|\; \sigma \models \phi(P) \wedge \bigwedge_{i=1,n} x_i = \mathcal{A}(\tau_{<i})\right\}\right| \\
&= \left|\left\{\sigma\restriction_{\mathbf{Out}(P)} \;\middle|\; \sigma \models \phi(P) \wedge \bigwedge_{i=1,n} x_i = a_i\right\}\right| \\
&= \left|\left\{\sigma\restriction_{\mathbf{Out}(P)} \;\middle|\; \sigma \models \phi(P)[x_i \leftarrow a_i]\right\}\right| \\
&= \left|\mathcal{I}^{\mathbf{Out}(P)}_{\phi(P)}(x_{\mathcal{A}})\right| \\
&= \|\phi(P)(x_{\mathcal{A}})\|_{\mathbf{Out}(P)}
\end{aligned}
$$

□

Example 6.2.1 (continuing from p. 70). The Max#SAT problem corresponding a leakage objective with regard to P is the following:

$$\mathsf{M}x_2, x_4.\ \mathsf{Я}x_1, x_3.\ \exists y_1, y_2.\ (x_1 \Leftrightarrow y_1 \vee y_2) \wedge (x_3 \Leftrightarrow x_2 \wedge y_2)$$

In this case an optimal answer is $x^\star = \{x_2 \mapsto \top, x_4 \mapsto \top\}$, which yields the following optimal non-adaptive attack:

$$\mathcal{A}^\star(\tau) = \top$$

Chapter 7

Max#SAT resolution

We proved in Chapter 6 that this Max#SAT is of key importance when studying the security of programs in the context of non-adaptive attackers. In this chapter, we propose a new resolution algorithm for Max#SAT.

Disclaimer Part of the results mentioned in this chapter are also exposed in papers by the author[Vig+22].

As mentioned earlier #SAT is the problem of counting the solutions of a quantifier-free propositional formula, the counting version of the SAT problem. Max#SAT is the problem of optimizing, according to some propositional variables, the number of solutions according to the others. We generalize this problem to allow an existential prefix in the formula.

Unfortunately, Max#SAT has high complexity [Tor91], and practical solving methods remain costly and this motivates the use of approximation schemes in order to solve the problem in reasonable time and space, while providing guarantees on the result.

7.1 Preliminary properties of Max#SAT

In this section, we prove some important properties with regard to Max#SAT, useful in order to realize the structure that arises from the problem. What we mean by that is that while this combinatorial optimization problem is computationally hard to solve, the problem obeys some laws that one can exploit in order to solve it. These properties were also exploited in other papers on the subject [ALM22; Aud+22].

Lemma 7.1 (Monotony). *Given a propositional formula $\phi(X, Y, Z)$, $A \subseteq B \subseteq X$ and $x \in \mathcal{M}_X(\phi)$, the count of partial witnesses is monotone:*

$$\|\phi(x\downharpoonright_B)\|_Y \leq \|\phi(x\downharpoonright_A)\|_Y$$

Proof. First, following Definition 2.3 we have:

$$[x]_B \subseteq [x]_A$$

Hence, following Definition 6.2:

$$\mathcal{I}^Y_\phi(x\downharpoonright_B) \subseteq \mathcal{I}^Y_\phi(x\downharpoonright_A)$$

The result follows. □

This lemma leads to a first interesting property, stating that the count of a complete witness is bounded by the count of the partial witness of which it is an extension. This relation is a key to the application of CEGAR methods which will be applied in this chapter, as it allows to quickly overestimate the counts of many complete witnesses by the count of an equivalence class that contains them.

Corollary 7.2. *Given a formula $\phi(X, Y, Z)$, the count of a partial witness is an upper-bound of the count of its extensions:*

$$\forall x' \in [x]_E, \left\|\phi\big(x'\big)\right\|_Y \leq \|\phi(x\downharpoonright_E)\|_Y$$

We continue by proving two useful properties about how the count of a given witness evolves when changing the formula of interest in some way. The objective behind these properties is to provide the building blocks required to devise the resolution algorithm for Max#SAT.

Property 7.3. *For a given $\phi(X, Y, Z)$ and $\phi'(X, Y, Z)$ such that $\phi \models \phi'$, a witness x and $E \subseteq X$ we have:*

$$\|\phi(x\downharpoonright_E)\|_Y \leq \left\|\phi'(x\downharpoonright_E)\right\|_Y$$

Proof. For any $y \in \mathcal{I}^Y_\phi(x\downharpoonright_E)$, we have:

$$\begin{aligned} y &\models \phi(x\downharpoonright_E) \\ &\models \phi'(x\downharpoonright_E) \end{aligned}$$

We get $y \in \mathcal{I}^Y_{\phi'}(x\downharpoonright_E)$ and hence $\mathcal{I}^Y_\phi(x\downharpoonright_E) \subseteq \mathcal{I}^Y_{\phi'}(x\downharpoonright_E)$ and the result follows. □

Property 7.4. *Given $\phi(X, Y, Z)$, $\psi(X)$ and $x \models \psi$, we have:*

$$\|\phi(x)\|_Y = \|(\phi \land \psi)(x)\|_Y$$

Proof. Since $x \models \psi$ and ψ does not depend on Y and Z, $(x, y, z) \models \phi$ if and only if $(x, y, z) \models \phi \land \psi$. □

7.2 `BaxMC`: a CEGAR algorithm for Max#SAT

This section exposes one of the contribution of this thesis: a new algorithm to solve the Max#SAT problem based on CEGAR methods.

Roughly speaking, this algorithm consists in iterating over possible *witnesses* x of ϕ. If the count for x is less than the current best solution, it *blocks a generalization* of x such that all extensions of this generalization are *worse* than the current best solution (Lines 14 and 16). It hence removes a chunk of the search space at each iteration. Otherwise, it saves the candidate, which is then the new maximum and blocks it (Lines 10 and 16), removing only one candidate from the search space.

Together with the formula, the algorithm takes multiple precision parameters:

- (ϵ_i) that are called *tolerance* parameters [CMV13];
- (δ_i) that are called *confidence* parameters [CMV13];

- κ that is called the *persistence* parameter.

As we will see in the correctness proofs of Algorithm 1, these parameters will influence the guarantees that one can provide about the returned answer. Informally, tolerance parameters influence the distance between the returned answer and the actual maximal value; the greater the tolerance parameter, the wider the set of acceptable values. Confidence parameters impact the probability that the returned answer is inside the tolerance bounds: the greater the confidence to one, the greater the probability is that the count is in the tolerance interval.

Algorithm 1 Pseudocode for the `BaxMC` algorithm

```
1:  function BaxMC_{$\epsilon_0,\epsilon_1,\delta_0,\delta_1,\delta_2,\kappa$}($\phi(X, Y, Z)$)
2:      $\phi_s \leftarrow \phi$
3:      $x_m \leftarrow \top$
4:      $n_m \leftarrow 0$
5:      $N \leftarrow MC^Y_{\epsilon_0,\delta_0}(\phi(\emptyset))$
6:      while $n_m < \frac{N}{1+\kappa}$ do
7:          $x \xleftarrow{\$} \mathcal{M}_X(\phi_s)$                    ▷ Pick a new candidate
8:          $c \leftarrow MC^Y_{\epsilon_1,\delta_1}(\phi_s(x))$
9:          if $c > n_m$ then                                   ▷ New maximum
10:             $x_m \leftarrow x$
11:             $n_m \leftarrow c$
12:             $E \leftarrow X$
13:         else                                                ▷ Find generalization
14:             $E \leftarrow$ Generalize$_{\delta_2}(x, \phi_s, n_m)$
15:         end if
16:         $\phi_s \leftarrow \phi_s \wedge \neg(x\downharpoonleft_E)$                         ▷ Block
17:         $N \leftarrow MC^Y_{\epsilon_0,\delta_0}(\phi_s(\emptyset))$
18:     end while
19:     return $x_m, n_m$
20: end function
```

The key step in the algorithm is the generalization step. Indeed, because of Corollary 7.2, this generalization step turns into the problem of finding, using the least number of calls to the model counting oracle, the smallest partial assignment such that its count is below the current maximum.

For the correctness of the algorithm, it is sufficient to assume that the Generalize function satisfies some properties of correctness. These correctness properties are the one mentioned above: the Generalize algorithm has to return sets such that the partial assignment on this set of the current candidate has a count smaller than the current maximum.

Definition 7.1 (n-bounding)**.** Given $\phi(X, Y, Z)$, $x \in \mathcal{M}_X(\phi)$ we say that $E \subseteq X$ is n-bounding if:

$$\|\phi(x\downharpoonleft_E)\|_Y \leq n$$

Definition 7.2 (Probably n-bounding)**.** Given $\phi(X, Y, Z)$, $x \in \mathcal{M}_X(\phi)$ we say that $E \subseteq X$ is *probably* n-bounding with confidence $1 - \delta$ when:

$$\mathbb{P}[\|\phi(x\downharpoonleft_E)\|_Y \leq n] \geq 1 - \delta$$

Definitions 7.1 and 7.2 are the direct translations of the correctness properties mentioned above in the exact and approximate cases. We present in this chapter an algorithm that satisfied these correctness requirements, as well as compare our algorithm to related works on the subject.

7.2.1 Termination and correctness with an exact #SAT oracle

In this section, each i-indexed variable of the algorithm denotes its value at the end of the i-th iteration of the main loop (Line 18). As we consider the exact version of the algorithm, all precision parameters are assumed to be equal to 0 and all calls to $MC^Y_{0,0}(\phi(x))$ return $\|\phi(x)\|_Y$. Furthermore, we assume that the confidence, tolerance and persistance parameters are all 0.

Theorem 7.5 (Termination with an exact #SAT oracle). *Algorithm 1 always terminates.*

Proof. By construction of $(\phi_{si})_i$ we have:

$$\forall i > 0.\ 0 \leq \|\mathcal{M}_X(\phi_{si+1})\| < \|\mathcal{M}_X(\phi_{si})\|$$

The sequence $(n_{mi})_i$ is obviously increasing. Again by construction of $(\phi_{si})_i$ and from Property 7.3, the sequence $(N_i)_i$ is decreasing and hence $(N_i - n_{mi})_i$ is decreasing.

Putting all this together, $(\|\mathcal{M}_X(\phi_{si})\| + (N_i - n_{mi}))_i$ is strictly decreasing.

One can easily see that whenever $\|\mathcal{M}_X(\phi_{si})\| = 0$ it follows that $N_i = 0$ and $N_i - n_{mi} \leq 0$. Hence in all cases, after some iteration k, $N_k - n_{mk} \leq 0$ and the termination follows. □

Remark 7.2.1. The worst case complexity of Algorithm 1 is reached when it iterates over all the witnesses of the formula, i.e. $E_i = X$ for all i. This is equivalent to not doing generalization at all. In this case the algorithm performs an exponential number of calls to the #SAT oracle.

This is not a limitation: as Max#SAT is $\mathsf{NP}^{\mathsf{PP}}$ it is expected that an algorithm solving the problem exactly would make a number of model counting call exponential in the number of maximizing variables.

Let k be the number of iterations performed when Algorithm 1 terminates, then we have $n_{mk} \geq N_k$.

Lemma 7.6. *At every iteration i of Algorithm 1, we have:*

$$\mathcal{M}_X(\phi_{si}) = \mathcal{M}_X(\phi) - \bigcup_{j<i} [x_i]_{E_i}$$

Proof. This follows by construction of ϕ_{si}. □

Lemma 7.7. *At every iteration i of Algorithm 1, and assuming* Generalize *returns n-bounding generalizations of x (c.f. Definition 7.1) we have:*

$$\forall x' \in \bigcup_{j \leq i} [x_j]_{E_j}.\ \|\phi(x')\|_Y \leq n_{mi}$$

Proof. Let $j \leq i$, and let $x \in [x_j]_{E_j}$, by assumption and following Definition 7.1, we have $\left\|\phi_{sj}(x)\right\|_Y \leq n_{mj}$.

Then by construction of ϕ_{si} and Property 7.4 we have $\|\phi(x)\|_Y \leq n_{mj}$ which, as $(n_{mi})_i$ is increasing, proves the lemma. □

Theorem 7.8 (Correctness with an exact #SAT oracle). *Assuming that the* Generalize *function is n_m-bounding, Algorithm 1 is correct i.e., the returned tuple (x_m, n_m) satisfies:*

$$n_m = \|\phi(x_m)\|_Y = \max_{x \in \mathcal{M}_X(\phi)} \|\phi(x)\|_Y$$

Proof. Following Corollary 7.2 and since $n_{mk} \geq N_k$ we have:

$$\forall x \in \mathcal{M}_X(\phi_{sk}).\ \|\phi(x)\|_Y \leq N_k \leq n_{mk}$$

Then instantiating Lemma 7.7 at iteration k we have:

$$\forall x \in \bigcup_{i \leq k} [x_i]_{E_i}.\ \|\phi(x)\|_Y \leq n_{mk}$$

Following Lemma 7.6, at iteration k we have $\mathcal{M}_X(\phi) = \mathcal{M}_X(\phi_{sk}) \cup \bigcup_{i \leq k} [x_i]_{E_i}$, and the result follows. □

7.2.2 Correctness with a probabilistic #SAT oracle

Since the termination can be proven in the same way as in the exact case, we only prove the correctness.

Let us first recall the expected guarantees provided by an approximate model counter [CMV13], where the ϵ parameter characterizes the precision of the result and the δ parameter determines its associated confidence.

Property 7.9 (Correctness of the Model Counting)**.** *The count $MC^Y_{\epsilon,\delta}(\phi(x))$ returned by an approximate model counter satisfies the following:*

$$\mathbb{P}\left[\frac{1}{1+\epsilon} \leq \frac{MC^Y_{\epsilon,\delta}(\phi(x))}{\|\phi(x)\|_Y} \leq 1+\epsilon\right] \geq 1-\delta$$

These guarantees extend to partial witnesses naturally, i.e., queries of the form $MC^Y_{\epsilon,\delta}(\phi(x\lfloor_E))$.

The next theorem proves the correctness of Algorithm 1 in the approximate case and gives the associated *tight* bounds.

Theorem 7.10. *Let (x_m, n_m) be the result returned by the call* $\textsc{BaxMC}_{\epsilon_0,\epsilon_1,\delta_0,\delta_1,\delta_2,\kappa}(\phi(X, Y, Z))$, *and let*

$$M = \max_{x \in \mathcal{M}_X(\phi)} \|\phi(x)\|_Y$$

If the $\textsc{Generalize}_{\delta_2}$ *function is probably n_m-bounding with confidence $1 - \delta_2$ then:*

$$\mathbb{P}\left[\frac{1}{1+\epsilon_1} \leq \frac{n_m}{\|\phi(x_m)\|_Y} \leq 1 + \epsilon_1\right] \geq 1 - \delta_1$$

and

$$\mathbb{P}\left[\|\phi(x_m)\|_Y \geq \frac{M}{(1+\epsilon_0) * (1+\epsilon_1) * (1+\kappa)}\right] \geq (1 - \delta_1) * \min(1 - \delta_2, 1 - \delta_0)$$

Proof. Let ϕ_S be the final value of the variable ϕ_s after the last iteration of the **while** loop. We have the following guarantees from the approximate model counter (Property 7.9):

$$\mathbb{P}\left[\frac{1}{1+\epsilon_1} \leq \frac{n_m}{\|\phi(x_m)\|_Y} \leq 1 + \epsilon_1\right] \geq 1 - \delta_1 \tag{7.1}$$

$$\mathbb{P}\left[\frac{1}{1+\epsilon_0} \leq \frac{N}{|\mathcal{M}_Y(\phi_S)|} \leq 1 + \epsilon_0\right] \geq 1 - \delta_0 \tag{7.2}$$

Assuming that the $\textsc{Generalize}_{\delta_2}$ function is probably n_m-bounding with confidence $1-\delta_2$, we also have that for any $x \in \mathcal{M}_X(\phi \wedge \neg\phi_S)$ it holds:

$$\mathbb{P}[\|\phi(x)\|_Y \leq n_m] \geq 1 - \delta_2. \tag{7.3}$$

After the last iteration of the **while** loop (Line 6) we have that $n_m * (1 + \kappa) \geq N$. Using this and Equation (7.2) and Property 7.4 we get that for any $x \in \mathcal{M}_X(\phi_S)$ it holds

$$\begin{aligned}
\mathbb{P}[\|\phi(x)\|_Y \leq n_m * (1+\kappa) * (1+\epsilon_0)] &\geq \mathbb{P}[\|\phi(x)\|_Y \leq N * (1+\epsilon_0)] \\
&= \mathbb{P}[\|\phi_S(x)\|_Y \leq N * (1+\epsilon_0)] \\
&\geq \mathbb{P}[|\mathcal{M}_Y(\phi_S)| \leq N * (1+\epsilon_0)] \\
&\geq 1 - \delta_0.
\end{aligned}$$

From Equation (7.3), for any $x \in \mathcal{M}_X(\phi \wedge \neg\phi_S)$ it holds

$$\begin{aligned}
\mathbb{P}[\|\phi(x)\|_Y \leq n_m * (1+\kappa) * (1+\epsilon_0)] &\geq \mathbb{P}[\|\phi(x)\|_Y \leq n_m] \\
&\geq 1 - \delta_2.
\end{aligned}$$

Hence, for any $x \in \mathcal{M}_X(\phi)$ it holds

$$\mathbb{P}[\|\phi(x)\|_Y \leq n_m * (1+\kappa) * (1+\epsilon_0)] \geq \min(1 - \delta_2, 1 - \delta_0)$$

and hence

$$\mathbb{P}\left[\frac{n_m}{1+\epsilon_1} \geq \frac{M}{(1+\kappa) * (1+\epsilon_0) * (1+\epsilon_1)}\right] \geq \min(1 - \delta_2, 1 - \delta_0) \tag{7.4}$$

Combining this with the Equation (7.1), we obtain

$$\mathbb{P}\left[\|\phi(x_m)\|_Y \geq \frac{M}{(1+\kappa)*(1+\epsilon_0)*(1+\epsilon_1)}\right] \geq (1-\delta_1)*\min(1-\delta_2, 1-\delta_0)$$

□

The following corollary instantiates Theorem 7.10 in order to get the standard *probably approximately correct* form (as in Property 7.9).

Corollary 7.11. *For any $0 < \epsilon, \delta < 1$, if in the call of the BaxMC function, we take as parameters $\epsilon_0 = \epsilon_1 = \kappa = \sqrt[3]{1+\epsilon} - 1$, $\delta_0 = \delta_2 = \frac{\delta}{2}$ and $\delta_1 = \frac{\delta}{2*(|X|+1)}$, then the result (x_m, n_m) satisfies the following inequalities:*

$$\mathbb{P}\left[\frac{1}{1+\epsilon} \leq \frac{n_m}{\|\phi(x_m)\|_Y} \leq 1+\epsilon\right] \geq 1-\delta$$

$$\mathbb{P}\left[\|\phi(x_m)\|_Y \geq \frac{M}{1+\epsilon}\right] \geq 1-\delta$$

where

$$M = \max_{x \in \mathcal{M}_X(\phi)} \|\phi(x)\|_Y$$

Proof. It is easy to check that $\epsilon_1 = \sqrt[3]{1+\epsilon} - 1 \leq \epsilon$, $\delta_1 = \frac{\delta}{2*(|X|+1)} < \delta_2 = \frac{\delta}{2} < \delta$, $(1+\epsilon_0)^3 = 1+\epsilon$ and $(1-\delta_1)*(1-\delta_0) = 1-\delta_0-\delta_1+\delta_0*\delta_1 > 1-2*\delta_0 = 1-\delta$. □

7.3 Generalizing witnesses in BaxMC

In this section we focus on process of finding an n-bounding set (c.f. Definition 7.1) given a witness. As mentioned before, finding smaller n-bounding sets in a small number of #SAT oracles is of key importance in our algorithm, as it allows skipping big parts of the witness set at each iteration of Algorithm 1, and thus avoid many calls to the #SAT oracle.

We thus focus on the search for *minimal* n-bounding sets, meaning that any subset of said set is not n-bounding anymore. It is important to note that because of the Monotony lemma (Lemma 7.1), n-boundingness is a monotone property: any superset of an n-bounding set is itself n-bounding. This property turns the problem of searching for minimal n-bounding sets into an instance of *Minimal Set subject to a Monotone Predicate* (MSMP) [MJB13].

We first present in this section a custom linear sweep algorithm, augmented with heuristics that helps it scale in our setting. We then discuss other algorithms in the field.

7.3.1 Linear sweep algorithm

Algorithm 2 is a linear sweep algorithm that solves the instance of the MSMP problem in our setting. It heavily exploits the fact that *relaxing* a literal in a witness can only increase the count of said witness, but fortunately, we can bound this increase. This is formally set in Property 7.12

Algorithm 2 Pseudocode for the generalization algorithm

```
function GENERALIZE_δ($x$, $\phi(X,Y,Z)$, $n_m$)
    $E \leftarrow X$

    for all $X_i \in X$ do                                          ▷ Step 1: Redundancy elimination
        if $\phi(x[X_i \leftarrow \neg x(X_i)], Y, Z)$ UNSAT then
            $E \leftarrow E - \{X_i\}$
        end if
    end for

    $k \leftarrow \log n_m - \log MC^{x}_{Z,\epsilon}(E(\phi))\delta_1$
    while $k > 0 \wedge |E| > 0$ do                                  ▷ Step 2: Log-elimination
        $A_k \overset{\$}{\leftarrow} \{V \subseteq E \mid |V| = k\}$
        $c \leftarrow MC^{Y}_{\epsilon,\delta_1}(\phi(x|_{E-A_k}))$
        if $c \leq \frac{n_m}{1+\epsilon}$ then
            $E \leftarrow E - A_k$
            $k \leftarrow \log n_m - \log c$
        else
            $k \leftarrow k - 1$
        end if
    end while

    for all $X_i \in E$ do                                          ▷ Step 3: Refinement
        if $MC^{Y}_{\epsilon,\delta_1}\Big(\phi\Big(x\downharpoonright_{E-\{X_i\}}\Big)\Big) \leq \frac{n_m}{1+\epsilon}$ then
            $E \leftarrow E - \{X_i\}$
        end if
    end for

    return $E$
end function
```

Property 7.12. *Given a propositional formula $\phi(X,Y,Z)$ such that $x \in \mathcal{M}_X(\phi)$, $E \subseteq X$ and $X_i \in X$, we have:*

$$\|\phi(x\downharpoonright_E)\|_Y \leq \Big\|\phi\Big(x\downharpoonright_{E-\{X_i\}}\Big)\Big\|_Y \leq \|\phi(x\downharpoonright_E)\|_Y + \|\phi(x\downharpoonright_E[X_i \leftarrow \neg x(X_i)])\|_Y$$

Proof. The first inequality is a direct consequence of Lemma 7.1. The last inequality follows from Definition 6.2 and the triangle inequality. □

For efficiency reasons, the steps mentioned in Algorithm 2 are in a precise order. The reason behind this is:

1. The first step relies on a consequence of Property 7.12, allowing relaxing variables with simple calls to a SAT solver.

2. The *log-elimination* generalization is a heuristic allowing to do *big steps* in the generalization process by relaxing multiple variables at each loop turn.

3. The *refinement* pass generalizes x in such a way that the returned set is minimal, i.e. that none of the further generalizations of the returned value satisfies Definition 7.1.

The returned E is guaranteed only to be a *local minimum* (with regard to the cardinality of the set), it may not be the *smallest* set such that Definition 7.1 holds. This is because of order in which we consider variables of X in Algorithm 2 matters, and thus different variable orders may lead to different minimal subsets. This in turns opens the door to the use of heuristics during variable scheduling.

7.3.2 Correctness and complexity with an exact #SAT oracle

Let us prove the correctness of Algorithm 2 in the context of an exact #SAT oracle. This will complete the correctness proof of Algorithm 1 started in Section 7.2.1.

Property 7.13 is a direct consequence and ensures the correctness of the first step in the generalization algorithm. Informally, this allows removing *redundant* literals within a witness, while preserving its count.

Property 7.13. *If $\phi(x\downharpoonleft_E[X_i \leftarrow \neg x(X_i)], Y, Z)$ is not satisfiable then:*

$$\|\phi(x\downharpoonleft_E)\|_Y = \left\|\phi\Big(x\downharpoonleft_{E-\{X_i\}}\Big)\right\|_Y$$

Proof. This follows directly from Property 7.12. □

Theorem 7.14. *Algorithm 2 terminates and is correct: the returned set E satisfies Definition 7.1:*

$$\|\phi(x\downharpoonleft_E)\|_Y \leq n$$

Proof. In the **while** loop at Line 11 we can see that, at each iteration, either $|E|$ or k decreases. All other loops have a bounded number of iterations, thus ensuring the termination of the algorithm.

During any update of the temporary value E (Lines 6, 15 and 24), we ensure that the new value of E satisfies Definition 7.1:

1. At Line 6, Property 7.13 keeps the model counting stable.

2. At Lines 15 and 24, the update is guarded by the explicit check of the property (in the **if** statements at Lines 14 and 23).

□

7.3.3 Correctness with an approximate #SAT oracle

Theorem 7.15. *Let $E \subseteq X$ be the set returned by the call* $\textsc{Generalize}_\delta(x, \phi(X,Y,Z), n)$*, and assume that*

$$\mathbb{P}[\|\phi(x)\|_Y \leq n] \geq 1 - \frac{\delta}{|X|+1}$$

Then:

$$\mathbb{P}[\|\phi(x\downharpoonleft_E)\|_Y \leq n] \geq 1 - \delta$$

Proof. Using Property 7.13, the variable E after the first loop within Algorithm 2 satisfies $\|\phi(x)\|_Y = \|\phi(x\downharpoonleft_E)\|_Y$.

We denote by $C_{x\downharpoonleft_V}$ the value returned by the call $MC^Y_{\epsilon,\delta_1}(\phi(x\downharpoonleft_V))$. Since each time we update E to a set V we ensure $C_{x\downharpoonleft_V} \leq \frac{n}{1+\epsilon}$, we have the following probability:

$$\mathbb{P}[\|\phi(x\downharpoonleft_E)\|_Y \leq n] \geq 1 - \delta_1$$

Let E_l denote the value obtained after l updates of variable E during $\textsc{Log-elimination}$ and $\textsc{Refinement}$ steps within Algorithm 2 and let us denote by P_l the confidence associated with the set E_l being probably n-bounding.

Using that we update E to the value E_l only if $C_{x\downharpoonleft_{E_l}} * (1+\epsilon) \leq n$, we have the following recursive relation:

$$\begin{aligned} P_l &= \mathbb{P}\big[\big\|\phi(x\downharpoonleft_{E_l})\big\|_Y \leq n\big] * P_{l-1} \\ &\geq (1-\delta_1) * P_{l-1} \\ &\geq (1-\delta_1)^l * P_0 \\ &\geq (1-\delta_1)^l * \left(1 - \frac{\delta}{|X|+1}\right) \end{aligned}$$

Thus, as $l \leq |X|$, if we take $\delta_1 = \frac{\delta}{|X|+1}$ and we call the #SAT oracle with parameters $\left(\epsilon, \frac{\delta}{|X|+1}\right)$ we get:

$$\begin{aligned} \mathbb{P}[\|\phi(x\downharpoonleft_E)\|_Y \leq n] &\geq (1-\delta_1)^{l+1} \\ &\geq 1 - (l+1) * \delta_1 \\ &\geq 1 - \delta \end{aligned}$$

□

Remark 7.3.1. The bound with respect to the number of updates is tight. The worst case is reached when the only valid subset of X is X itself, that is when the model cannot be generalized.

7.4 Breaking symmetries in Max#SAT

Symmetries are a special kind of permutations of the input variables of a formula. Exploiting or breaking symmetries in SAT formulas has long been a topic of interest (for a survey, see [Zha21]).

For instance, if a formula is symmetric with respect to some permutations of variables then for each blocking clause C, the solver may need to generate the full orbit of C by the group of permutations, leading to combinatorial explosion. Breaking the symmetry means selecting one solution per orbit by adding a predicate called *symmetry breaking predicate* to the formula, purposefully generated to break the symmetries. The resulting formula is equisatisfiable (it has a solution if and only if the original one has a solution), but often simpler to solve.

7.4.1 Correctness in the presence of symmetries

In our context, handling symmetries within the witness set reduces the size of the search space, lowering the complexity. We give in this section arguments about why this is true.

Definition 7.3. Given a propositional formula $\phi(X, Y, Z)$, a *symmetry* of ϕ is a bijective function $\sigma : \overline{X} \mapsto \overline{X}$ that preserves negation, that is $\sigma(\neg X) = \neg\sigma(X)$, and such that, when σ is lifted to formulas, $\sigma(\phi) = \phi$ up to commutativity of $\wedge$ and $\vee$.

S_ϕ denotes the set of all symmetries of ϕ. We lift S_ϕ to models by defining the set of symmetries of a model x, $S_\phi(x) = \{x \circ \sigma \mid \sigma \in S_\phi\}$.

Theorem 7.16. *In Algorithm 1, picking only one x per symmetry class of ϕ preserves the correctness of the algorithm both in the exact and approximate cases.*

Proof. Whatever the method used to select only one member of each symmetry class, this corresponds to creating a symmetry breaking predicate $\psi(X)$ and solving the problem over $\phi \wedge \psi$, and thus the Property 7.4 applies. $\square$

7.4.2 Implementing Max#SAT symmetry breaking

We detect symmetries in ϕ using the automorphisms of a colored graph representing the formula, defined as follows:

- For each variable, create two nodes: one for the positive literal, and one for the negative literal. Use color 0 if the variable is in X, otherwise use color 1. Add an edge (*Boolean consistency edge*) between the two nodes.
- For each clause, create a node, and assign to it the color 2. Add an edge between this clause node and every node corresponding to a literal present in the clause.

Many tools can be used in order to list the automorphisms of a graph. In our case, we used BLISS [JK07] because of its C++ interface, and its performance.

After detecting the symmetries, one can use any symmetry breaking technique available, either static [Dev+16] or dynamic [Met+18]. In our implementation, we chose to use CDCLSYM [Met+18] because of its ease of use by integrating it into Algorithm 1, and because it avoids generating complex symmetry breaking predicates ahead of time.

The implementation of Algorithm 1 is thus slightly changed: whenever we sample a new assignement x (Line 7) we perform the necessary checks as per [Met+18]. If the checks indicate that we are entering a symmetric subspace of models of the formula, we block that subspace and perform a new sampling (effectively going back to Line 7)

7.5 Heuristics and optimizations

We present in this section heuristics used in both Algorithms 1 and 2 in practice, and discuss their effectiveness.

7.5.1 Progressive construction of the candidate

A simple heuristic is to gradually add literals to the current candidate x in Algorithm 1 at Line 7. By realizing that the count of the partial candidate x is below to current maximum, this allows calling Generalize on a partial assignment instead of a complete one. This may decrease the number of calls to the #SAT oracle as it anticipates work that is done in Algorithm 2.

7.5.2 Leads

When performing the generalization in Algorithm 2, one can see that we can extract *hints* about promising parts of the search space when relaxing variables. Indeed, when relaxing parts of the witness, if the count of the generalization goes above n_m (Lines 18 and 25), then this part of the search space may contain an improvement over the current solution.

Following this intuition, one can hold a *sorted list*[1] of relaxations whose count is above the current best known maximum, and use it to favor parts of the search space that look promising. We call these promising relaxations *leads*. More formally, given $\tilde{x}\lfloor_E$ a lead, when searching for a new solution in Algorithm 1 at Line 7, instead of searching in $\mathcal{M}_X(\phi_s)$, one would search in $\mathcal{M}_X(\phi_s) \cap [\tilde{x}]_E$.

Let $L_n(\phi)$ denote the set of leads currently known to the solver with count lower than n. When the currently known maximum is improved in Algorithm 1 at Line 10, we can block all leads whose count is below the new maximum:

$$\phi_{s i+1} = \phi_{s i} \wedge \bigwedge_{[\tilde{x}]_E \in L_{n_m}(\phi)} \neg(\tilde{x}\lfloor_E)$$

7.5.3 Decision heuristic

As discussed in Section 7.3, the performances of the algorithm depend on the order in which variables are considered in various parts of the solving process (in the generalization and during the optimization presented in Section 7.5.1). One can see that this kind of problem, that we call *variable scheduling*, is actually predominant when solving SAT problems, and even #SAT problems [Mos+01; SBK05].

One first heuristic arises from the leads described in Section 7.5.2. One can use the leads list as indications for literals leading to promising parts of the search space, by finding the literal which appears the most in the leads. We call this heuristic `leads`.

[1]The order to use here is: first the count of the relaxation $\|\phi(\tilde{x}\lfloor_E)\|_Y$ then its size $|E|$, both ascending.

Another decision heuristic can be devised using VSIDS [Mos+01]. The idea is to assign a weight to each literal based on its last appearance in a blocking clause. The weight of each literal is recomputed at each blocking operation as follows:

- If the literal is part of the blocking clause, we multiply it's weight by some constant α
- Otherwise, we *decay* its weight by multiplying it by some constant β.

This heuristic showed promising results in both SAT and #SAT [SBK05]. We call this heuristic `vsids`.

One could also choose the next decision variable at random, which we call `rnd`. And finally, one could just pick the decision variables in the order they are provided to the tool (that is the order of their names), which we call `none`.

7.5.4 Handling equivalent literals

Equivalent literals are a notorious property of propositional formulas which, when exploited, results generally in better runtime performances [LMY21].

Definition 7.4. Given a propositional formula ϕ, we say that two literals $L_i \in \overline{V}$ and $L_j \in \overline{V}$ are *equivalent* if $\phi \models L_i \Leftrightarrow L_j$.

Equivalent literals allow to simplify formulas based on the following theorem.

Theorem 7.17. *Let ϕ be a propositional formula and two equivalent literals L_i and L_j. Then solving the* MAX#SAT *problem for ϕ is reduced to solving the* MAX#SAT *problem for the simpler formula ϕ' obtained by replacing all occurrences of L_j (resp. $\neg L_j$) by L_i (resp. $\neg L_i$) when:*

1. *either L_i and L_j are in the same literal class (either $\overline{X}$, $\overline{Y}$ or $\overline{Z}$)*
2. *or $L_i \in \overline{X}$ and $L_j \in \overline{Y} \cup \overline{Z}$*
3. *or $L_i \in \overline{Y}$ and $L_j \in \overline{Z}$.*

Theorem 7.17 can be repeated multiple times in order to further simplify the formula. Literal equivalence can be detected using binary implication graphs [HJB11].

7.5.5 Dichotomic search

As it has been remarked in other areas in the context of combinatorial optimization based on the enhancement of a better solution, one can actually search for a solution by dichotomy.

In this section, we call η the value of this dichotomic target. The actual value of η is not important as long as it respects $n_m \leq \eta \leq N$ (using the notations of Algorithm 1).

Within our context, there could be two cases:

1. Either a solution exists with a count greater than η, which we shall find.

2. Or there is no such solution and we would then have proved that there exist no solution x such that $\|\phi(x)\|_Y \geq \eta$.[2]

In order to take both these outcomes into account and use them in the solving process, we need to have the ability to retract blocking clauses added in Line 16 (Algorithm 1), as these would be computed against η, which may be invalidated in the future (case 2).

This is done be introducing, for each new η value, a *pivot* variable, in order to control the activation of the learned clauses during the solve process against said η. This pivot variables is denoted p_η from now on.

The idea is then, for each learned clause C, to transform it into $C \vee p_\eta$ and to use the formula $\phi \wedge \neg p_\eta$ for each model counting operation. Then, depending on the two cases mentioned above, one would learn a new unit clause: if a new maximum is found (resp. not found), one would learn $\neg p_\eta$ (resp. p_η) meaning that the learned clauses are accepted (resp. denied).

Furthermore, whenever no new maximum is found, we actually discovered a new bound on the maximum model counting of the remaining solutions in the search space. We can thus set $N = \eta$ and continue the algorithm from here.

[2]Note that in the approximate solving case, this overapproximation is then probabilistic.

Chapter 8

Experimental evaluation

Algorithm 1 has been implemented in an open-source tool written in C++ called `BaxMC`[1], including dynamic symmetry breaking techniques (Section 7.4) and all the heuristics discussed in Section 7.5. In this implementation, we only incorporated the approximate version of the algorithm using `ApproxMC5` [MA20] as an approximate model counting oracle and `CryptoMinisat` [SNC09] as a exact SAT solver oracle, using a technique similar to `CMSGen` [Gol+21] to sample the solutions within the witness space.

We use multiple benchmark sources, coming either from [FRS17][2], special stochastic SAT instances extracted from [LWJ18][3], or from MaxSat 2021 competition[4]. Benchmarks from this later class are transformed using the method from [FRS17]. On average, these example formulas contain hundreds of variables and thousands of clauses.

All experiments are run on a HPE DL380 machine with 56 cores and 512 GB of RAM running Ubuntu 22.04 LTS 64b, with a 2-hour timeout, a 10 GB memory limit and with parameters $\delta = 0.2$, $\epsilon = 0.8$.

When not specified, `BaxMC` uses its default heuristics, which are `(leads,rnd)`.

8.1 Comparison to other solvers

`MaxCount` [FRS17] is used as an off-the-shelf solver of the problem, with parameters corresponding to $\delta = 0.2$, $\epsilon = 0.8$. Note that these are not the parameters used in the experiments in [FRS17] and that we reimplemented `MaxCount` using newer oracles. We did this in order to fairly compare how `MaxCount` and `BaxMC` behave when both provide the same correctness guarantees and use the same oracles.

This section attempts to answer the following questions:

RQ.1 Is `BaxMC` an effective solver when compared to `MaxCount` ?

RQ.2 Is there any difference in the answers returned by these two solvers ?

Figure 8.1 already answers **RQ.1**: `BaxMC` offers overall better performance when ran on our set of benchmarks, allowing it to solve approximately twice as much instances compared

[1] Accessible online at `https://gricad-gitlab.univ-grenoble-alpes.fr/tava/baxmc`

[2] Extracted from `https://github.com/dfremont/maxcount`

[3] Accessed at `https://github.com/vuphan314/ssat`

[4] `https://maxsat-evaluations.github.io/2021/`

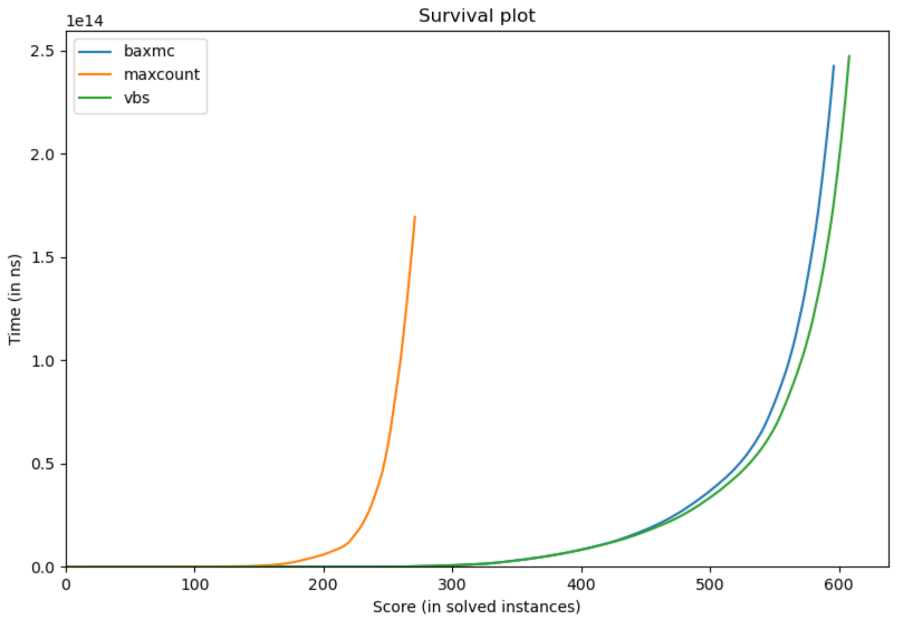


Figure 8.1: Performance comparison between `MaxCount` and `BaxMC`

to `MaxCount`. The difference between `BaxMC` and the virtual best solver (`vbs`) indicates that some benchmarks instances are beneficial to `MaxCount`.

We can complement the answer to **RQ.1** using Figure 8.2: some benchmarks are indeed solved faster by `MaxCount`, but the majority of the benchmarks are solved faster by `BaxMC`.

Figure 8.3 shows the mean squared error which is 1 minus the ratio to the best known answer. It is important to note that in this context, a time-out or memory-out is considered a bad answer and thus an error of 1. We say that a given solver got the *best answer* whenever the count of the witness it returns is greater than that of the others.

In this setting, we can clearly see that `BaxMC` gives overall better results than `MaxCount`. It is important though to note that `MaxCount` still gives better answers than `BaxMC` in some cases. This answers **RQ.2**.

8.2 Comparing heuristics influencing the search procedure

The *progressive construction* of the solution (Section 7.5.1) and the *dichotomic search* (Section 7.5.5) are two heuristics than can be put in a different category and compared independently as they influence mainly the search procedure of the algorithm.

We discuss in this section performance impact of the different combinations of these two heuristics.

We try to answer the following research questions:

RQ.3 Is the progressive construction of the solution benefic in terms of performance ?

RQ.4 Is the dichotomic heuristic benefic in terms of performance ?

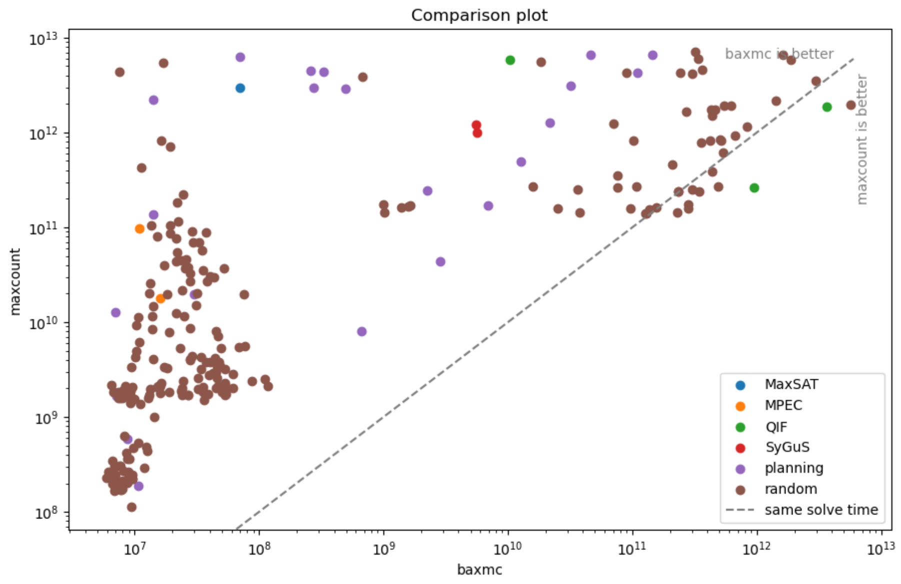


Figure 8.2: Comparison of runtimes (in ns) between `MaxCount` and `BaxMC`

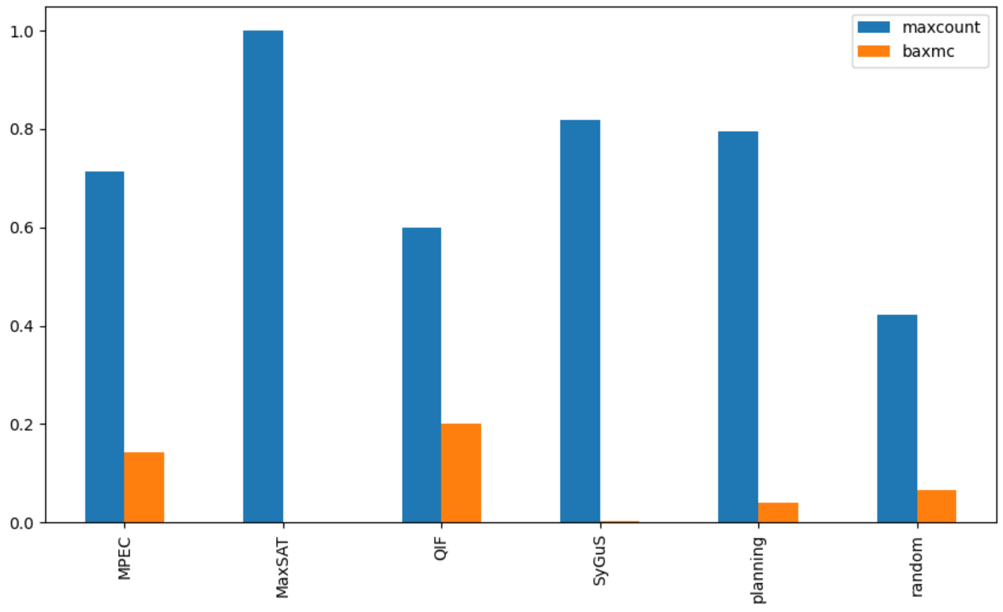


Figure 8.3: Answer comparison between `MaxCount` and `BaxMC`

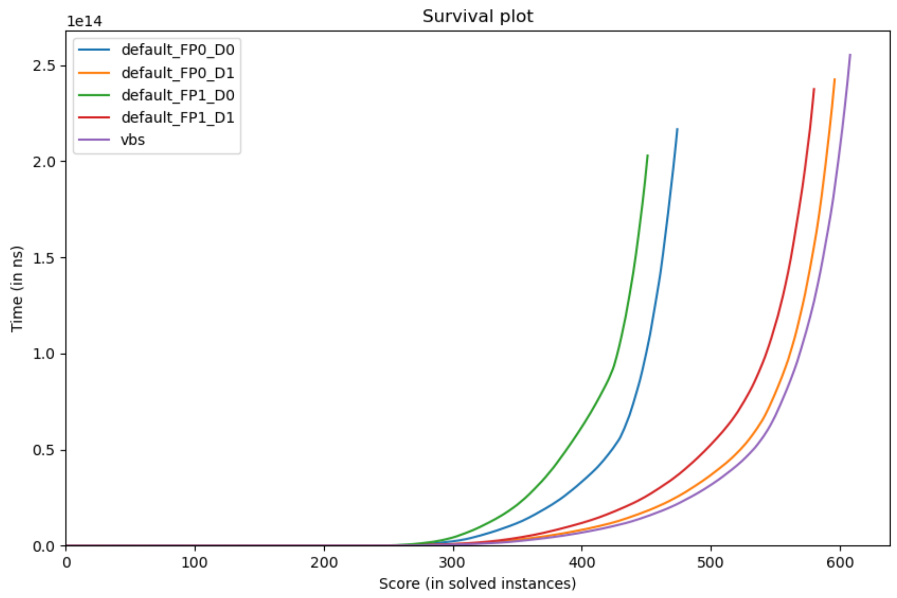


Figure 8.4: Performance comparison of the progressive construction and dichotomy heuristics. `D0/1` (resp. `FP0/1`) indicates whether the dichotomic heuristic (resp. progressive construction) is enabled.

RQ.5 In terms of answers, does one of the heuristics perform better ?

We can answer the first two questions from Figure 8.4 and especially by comparing each combination to `default_FP0_D0` which is just the default `BaxMC` configuration:

RQ.3 Progressive construction of the solution seems to hinder performance by a small amount.

RQ.4 Dichotomic search of a better solution seems to provide a consequent performance benefit.

Moreover, the difference between the VBS and the best performing solver indicates that this solver is not the best on all benchmarks. This naturally raises the question of the best solver for each benchmark and indicates some finer performance differences between heuristics comparisons.

Using Figure 8.5 we can realize that some benchmarks are still favorable to the progressive construction of the solution, but that most benchmarks favored by the progressive construction heuristic are actually benchmarks from the `random` set.

This yields another answer to **RQ.3** and **RQ.4**: while the performance gain of the dichotomic heuristic seems important, there are still cases where the progressive construction of the solution helps.

One can see in Figure 8.6 that the heuristic used to search for a solution matters with regard to the solution found. It seems though that the progressive construction of the solution (red and green) is worse in nearly all cases apart from some `planning` and `random` benchmark types. In contrast, the separation between enabling the dichotomic and non dichotomic

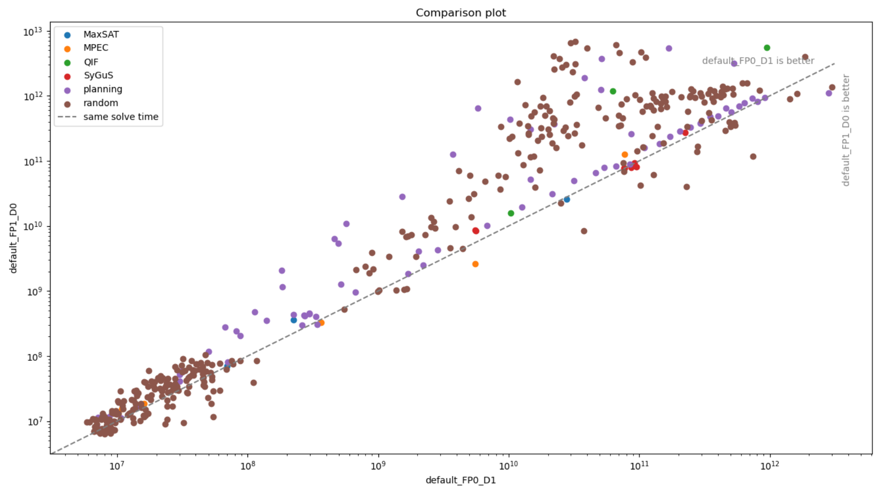


Figure 8.5: Comparison of runtimes (in ns) between the best and worst solver of Figure 8.4

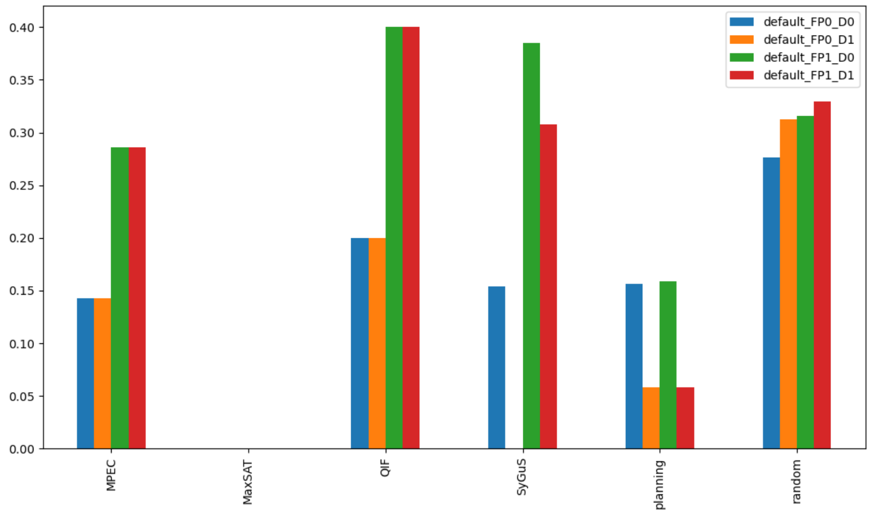


Figure 8.6: Answer comparison between heuristic combinations

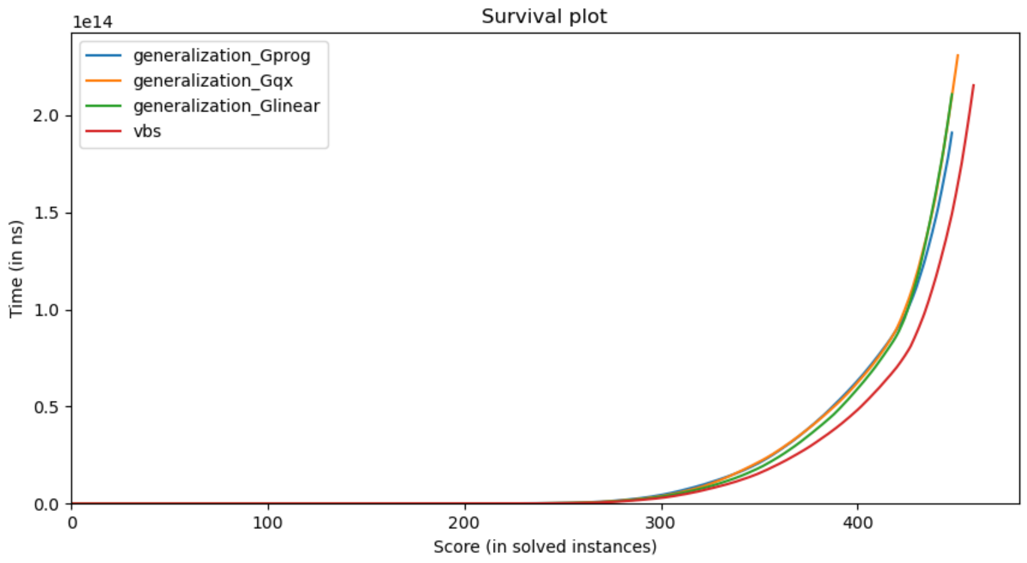


Figure 8.7: Performance comparison between generalization algorithms

resolution seems less clear. It is though important to note that the difference in answer quality is definitely present. This answers **RQ.5**: heuristics matter when solving and can yield different solutions, but the actual difference seems to depend on the instance source.

8.3 Comparing other generalization algorithms

As discussed in Section 7.3, the problem of generalizing a solution is an instance of the MSMP problem. The goal of this problem is to find, for a given set, a subset which is minimal with regard to inclusion and which satisfies a given predicate.

Using this, we give in this section an experimental evaluation of some generalization algorithms and compare them to Algorithm 2:

- `prog` is a progression-based algorithm taken from [MJB13].
- `qx` is the well known QuickXPlain algorithm [Jun04].
- `linear` is Algorithm 2.

Note that we do not provide proofs of correctness similar to Theorem 7.15 for `prog` and `qx` when used against an approximate model counting oracle. Because of the construction of these two algorithms, we think that their correctness proof should be similar to that of Theorem 7.15.

A survival plot of Algorithm 1 with the different algorithms is given in Figure 8.7. There does not appear to be a clear overall best generalization algorithm, but the difference between all the solvers and the virtual best solver indicates that the performances of a given generalization algorithm heavily depends on the benchmark. It is important to notice that the generalization process is still important: without it, Algorithm 1 degenerates into an exhaustive search for the optimal solution (see Remark 7.2.1). In order to enhance performances, `BaxMC` is currently implemented with the choice of the generalization algorithm left up to the user.

Notice that the end of the graph for `prog` is slightly lower than the others, indicating that the mean solving time of the instances is slightly smaller overall. Similarly, the height of the end of the graph of `qx` indicates a slightly longer mean solving time.

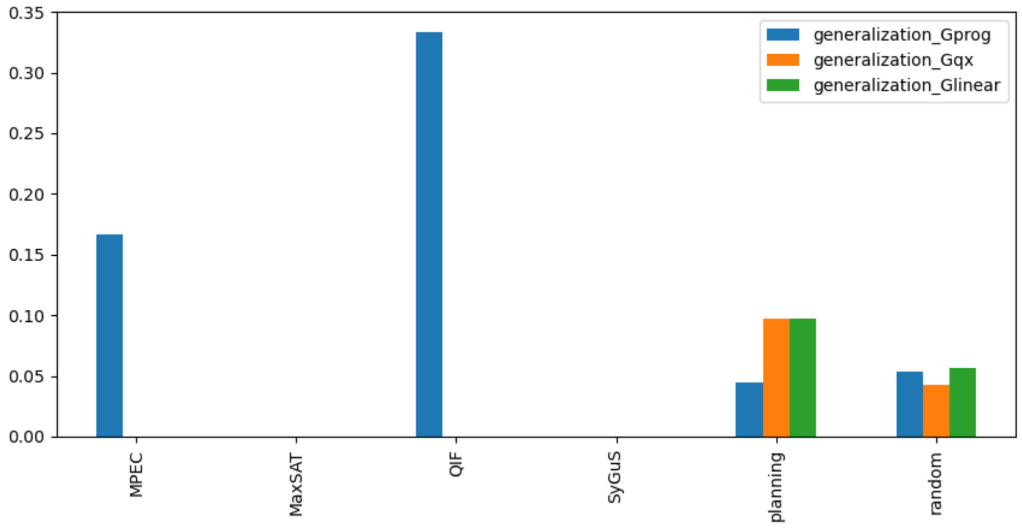


Figure 8.8: Answers comparison between generalization algorithms

Figure 8.8 shows the quality of the answers depending on the algorithm used. We can clearly see that `prog` gives overall slightly worse results, but that this depends on the kind of instance considered. Notice too that the difference in answer quality between `qx` and `linear` is small. This again indicates that having multiple generalization algorithms at hand is helpful in order to solve a given instance.

Part IV

Adaptive attacks

Chapter 9

Reducing to SSAT

After studying non-adaptive attacks in Part III, as well as the related counting problem, we study in this chapter the realm of adaptive attacks. Evidently, this is a strict generalization of non-adaptive attacks, and as we will see in this chapter, this is reflected by the corresponding counting problem. In this chapter, we prove that security evaluation against adaptive attackers can be reduced to the SSAT problem, following the same pattern as in Part III.

Let us recall first some facts about adaptive attackers in our context (i.e. normalized SIP programs). Given an attack $\mathcal{A}$, recall that Theorem 5.14 links the characteristic formula, a configuration (σ, τ) and the *partial* traces τ_i. This link involves the evaluation of $\mathcal{A}(\tau_{<i})$. Note that $\mathcal{A}(\tau_{<i})$ can be viewed as a function from $\mathbf{Val}^*$ to $\mathbf{Val}$: indeed, the length of the trace is now fixed, so the value of $\mathcal{A}(\tau_{<i})$ only depends on the values of the events along the trace. Let us denote these functions as $\mathcal{A}_i$.

We can now see that the problem of attack synthesis, when considering adaptive attackers, corresponds to that of the synthesis of the functions $\mathcal{A}_i$, and that one can construct the original function from its parts as follows:

$$\mathcal{A}(\tau) = \mathcal{A}_{\|\tau\|+1}(\tau)$$

Example 9.0.1. Let P be the following program:

```
x1 <- y1 ∨ y2
out(x1)
x2 <- in()
x3 <- x2 ∧ y2
out(x3)
x4 <- in()
```

The characteristic formula of P is:

$$\phi(P) = x_1 \Leftrightarrow (y_1 \vee y_2) \wedge x_3 \Leftrightarrow (x_2 \wedge y_2)$$

Let us consider the simple adaptive attack defined as follows:

$$\mathcal{A} \triangleq \begin{cases} x_1 & \text{if } \tau = [\text{out}(x_1)] \\ x_3 & \text{if } \tau = [\text{out}(x_1), \text{in}(x_2), \text{out}(x_3)] \end{cases}$$

In this case, we have the following two boolean functions:

$$\mathcal{A}_2(x_1) = x_1$$
$$\mathcal{A}_4(x_1, x_2, x_3) = x_3$$

The objective of this chapter is thus to define SSAT queries based on the SIP program. The answer of these queries should translate to the optimal attack with regard to the given objective. We will see that these constructions strictly generalize the constructions in Chapter 6.

9.1 Stochastic Satisfiability

We recall here the definition of SSAT, as well as refine some definitions in order to properly define the problem.

Definition 9.1 (Syntax of SSAT)**.** Given $\phi(x_1, \ldots, x_n)$ a boolean formula the syntax of SSAT is, with $Q_i \in \{\mathsf{M}, \mathsf{Я}\}$:

$$\Phi \triangleq Q_1 x_1. \ \ldots \ Q_n x_n. \ \phi$$

We denote the *matrix* of Φ as $\textbf{Mat}(\Phi) \triangleq \phi$, and we denote the *prefix* of Φ as $\textbf{Pref}(\Phi) \triangleq Q_1 x_1. \ \ldots \ Q_n x_n.$.

We also denote $\mathsf{Я}(\Phi)$ as the set of variables prefixed by $\mathsf{Я}$ in $\textbf{Pref}(\Phi)$. Similarly, $\mathsf{M}(\Phi)$ denotes the M-prefixed variables in $\textbf{Pref}(\Phi)$.

Interestingly enough, when introducing the problem, Papadimitriou considers games between two players, one being deterministic and the other probabilistic, and asks to find the optimal strategy for the deterministic attacker. This is very similar to our setting: (1) the deterministic player is the attacker; (2) the probabilistic player is the target program; (3) the optimal strategy is the optimal attack.

Now that we have defined the syntax of SSAT formulas, we can define their semantics. The objective is to compute the so-called *probability* of the formula, which is the density of models with regard to the set of all assignments.

Definition 9.2 (SSAT problem)**.** Given an SSAT formula Φ, we define the *probability* of Φ inductively as follows:

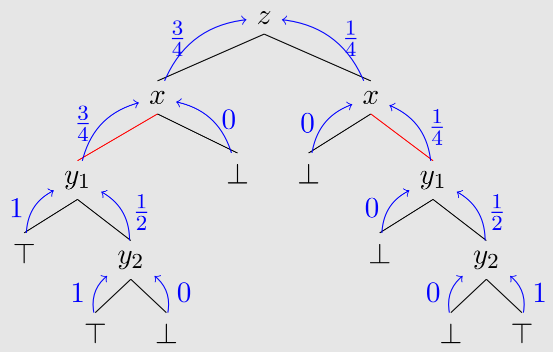
$$\textbf{Prob}(\mathsf{Я}y.\ \Phi(y)) = \frac{1}{2}\textbf{Prob}(\Phi(\top)) + \frac{1}{2}\textbf{Prob}(\Phi(\bot)) \qquad \textbf{Prob}(\top) = 1$$
$$\textbf{Prob}(\mathsf{M}x.\ \Phi(x)) = \max(\textbf{Prob}(\Phi(\top)), \textbf{Prob}(\Phi(\bot))) \qquad \textbf{Prob}(\bot) = 0$$

The SSAT problem asks to compute the probability of a given SSAT formula.

Example 9.1.1. Let Φ be the following SSAT problem:

$$\text{Я}z.\ \mathsf{M}x.\ \text{Я}y_1.\ \text{Я}y_2.\ (x \Leftrightarrow z) \wedge (z \Leftrightarrow y_1 \vee y_2)$$

We can unfold the tree of assignment for all variables. In the following, the left branch corresponds to assigning the parent to $\top$ and the right branch corresponds to assigning the parent to $\bot$, blue arrows are the probabilities computed recursively, and red edges denote the alternative selected by the maximum operator:



Thus $\textbf{Prob}(\Phi) = \frac{1}{2}$ by definition.

We can already see that the SSAT problem is a strict generalization problems presented before:

- SAT asks to determine whether $\textbf{Prob}(\mathsf{M}Z.\ \phi(Z)) = 1$
- #SAT asks to compute $2^{|Y|}\textbf{Prob}(\text{Я}Y.\ \phi(Y))$
- #∃SAT asks to compute $2^{|Y|}\textbf{Prob}(\text{Я}Y.\ \mathsf{M}Z.\ \phi(Y, Z))$
- MAX#SAT asks to compute $\textbf{Prob}(\mathsf{M}X.\ \text{Я}Y.\ \mathsf{M}Z.\ \phi(X, Y, Z))$

Remark 9.1.2 (A note on complexity). Interestingly enough, SSAT is of the same space complexity than the satisfiability of boolean formula augmented with universal and existential quantifiers (the so-called QBF problem), both being PSpace-complete.

This seems quite counter-intuitive: one asks about computing the number of models of a formula, while the other asks to prove that the formula is a tautology. This seems to indicate that SSAT is strictly harder that QBF.

This apparent gap between the two problems has showed by Toda in his famous theorem [Tod91], stating that any problem in the *polynomial hierarchy* (i.e. any QBF instance) can be solved in a polynomial number of calls to a #SAT oracle.

Theorem 9.1 (Toda's). $\mathsf{PH} \subseteq \mathsf{P}^{\#SAT}$

This indeed indicates the fundamentally different nature between these two classes of problems: while both of them can be solved in polynomial space, their time complexity is actually vastly different.

9.2 An alternative formulation: f-SSAT

Note that with the current definition of SSAT, the optimal strategy is recoverable by keeping track of the decisions taken at maximizing nodes when computing the probability. We thus provide an alternative (global) definition of SSAT that will be more convenient in our setting and that makes the strategy explicit. We also prove that this alternative definition is equivalent.

Definition 9.3 (f-SSAT strategy)**.** Given an SSAT formula Φ, a *strategy* s for Φ is a replacement from variables in $\mathsf{M}(\Phi)$ to expressions over $\mathsf{R}(\Phi)$ such that:

$$\forall x_i \in \mathsf{M}(\Phi).\ s(x_i) \in \mathcal{F}\left\langle \mathsf{R}(\Phi)_{<i} \right\rangle$$

We denote the set of strategies of Φ as **Strats**(Φ).

Informally, a strategy is a replacement *respecting the dependencies* of the formula. This means that the strategy corresponds to computing the values of every maximizing variable based on the values of the variables appearing before it in the prefix.

Definition 9.4 (Count of a strategy)**.** Given an SSAT formula Φ and $s \in$ **Strats**(Φ), we define the *count* of s as:

$$\|\Phi(s)\| \triangleq |\{y : \mathsf{R}(\Phi) \to \mathbb{B} \mid \mathbf{Mat}(\Phi)[x_i \leftarrow s(x_i)](y) \text{ is true}\}|$$

Definition 9.5 (f-SSAT problem)**.** Given an SSAT formula Φ, the f-SSAT problem asks to find $s^\star \in$ **Strats**(Φ) such that:

$$\|\Phi(s^\star)\| = \max_{s \in \mathbf{Strats}(\Phi)} \|\Phi(s)\|$$

Example 9.1.1 (continuing from p. 99). In this case, $\mathbf{Strats}(\Phi)$ is the set of all boolean functions with parameter z, that is $\mathbf{Strats}(\Phi) = \{\top, \bot, z_1, \neg z_1\}$. Each strategy corresponds to a selection of two branches in the second level of the assignment tree of the formula. In this case the optimal strategies correspond to the two branches selected by the max operator when computing the probability (in red).

Lemma 9.2 is easy to prove by definition of both problems. The only difference between the two problems is that one is so-called *witness-generating*, meaning that one has to also provide the strategy that maximizes the probability. This is of interest in our case.

Lemma 9.2 (Equivalence between SSAT and F-SSAT)**.** *Given an SSAT formula* Φ*, we have:*

$$\mathbf{Prob}(\Phi) = \frac{\max_{s \in \mathbf{Strats}(\Phi)} \|\Phi(s)\|}{2^{|\text{Я}(\Phi)|}}$$

9.3 Quantitative Reachability

Let us begin by defining the *SSAT prefix* associated with a SIP program. Our objective with the prefix construction is that this quantifier alternation follows the interaction alternation in the original program.

Definition 9.6 (Associated prefix)**.** Given a SIP program P in normal form, we define the *SSAT prefix* associated with P, denoted $\mathbf{Pref}(P)$, inductively as:

$$\begin{aligned}
\mathbf{Pref}(x_k \texttt{ <- in()}) &= \mathsf{M} x_k \\
\mathbf{Pref}(\texttt{out}(x_k)) &= \text{Я} x_k \\
\mathbf{Pref}(x \texttt{ <- } e) &= \epsilon \\
\mathbf{Pref}(P_1 \texttt{ ; } P_2) &= \mathbf{Pref}(P_1).\ \mathbf{Pref}(P_2)
\end{aligned}$$

The prefix is built in such way that: (i) the dependencies between inputs and outputs are preserved, (ii) the strategy generated by the F-SSAT query is a strategy regarding the inputs. Outputs need to be discussed: indeed, outputs can be thought as *existentially* quantified intuitively. Here we use *counting* quantifiers because, by construction of the characteristic formula of the program, output variables are *functionally dependent* on input and random variables. This means that once we fix the values for random and input variables, the value of the output variable is itself entirely determined.

Using the prefix associated to the program, we now define the main theorem of this section: in order to synthesize the optimal attack for a given quantitative reachability objective, one can use SSAT.

Note that the variables appearing in the prefix associated to a given program P are $\mathbf{Out}(P)$ and $\mathbf{In}(P)$. This means that when writing a SSAT query about P (using its characteristic formula) one needs to additionally specify how to handle $\mathbf{Rand}(P)$ and $\mathbf{Rest}(P)$, as these are sets of variables that appear in the characteristic formula but not in $\mathbf{Pref}(P)$.

Theorem 9.3 (Quantitative reachabilty to SSAT). *Given a* SIP *program P in normal form and a quantitative reachability objective e. The following are equivalent:*

- *The strategy $s_{\mathcal{A}^\star}$ is the optimal strategy for the SSAT query*

$$\mathbf{Pref}(P).\ \text{Я}\mathbf{Rand}(P).\ \mathsf{M}\mathbf{Rest}(P).\ \phi(P) \wedge e$$

- *The attack $\mathcal{A}^\star$, such that $\mathcal{A}_i^\star = s_{\mathcal{A}^\star}(x_i)$ is optimal with regard to the quantitative reachability objective e*

Proof. The proof follows closely the proof of the similar theorem in Chapter 6. Let $\mathcal{A}$ be an adaptive attack, i.e. such that $\mathcal{A}_i = s_{\mathcal{A}}(x_i)$, and let its associated SSAT strategy $s_{\mathcal{A}}$. Also, let $\Phi^e_{\mathcal{R}}(P)$ be the SSAT problem mentioned in the theorem.

Note that here again, per Property 4.2, it is sufficient to prove that:

$$|\{(\sigma,\tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) \mid \sigma(e) = \top\}| = \|\Phi^e_{\mathcal{R}}(P)(s_{\mathcal{A}})\|$$

Indeed, this in turns implies that the orders among the two sets (attacks and strategies) are the same, and thus that their maxima correspond.

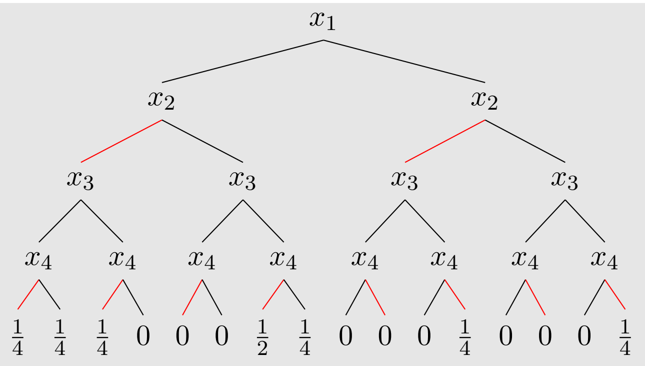
$$\begin{aligned}
|\{(\sigma,\tau) \in [\![P]\!]_{\mathcal{A}}(\mathbf{State}_0) \mid \sigma(e) = \top\}| &= \left|\left\{ (\sigma,\tau) \;\middle|\; \begin{array}{l} \tau = \sigma(\mathrm{tr}(P)) \\ \sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}(\tau_{<i}) \wedge e \end{array} \right\}\right| \\
&= \left|\left\{ \sigma \;\middle|\; \begin{array}{l} \tau = \sigma(\mathrm{tr}(P)) \\ \sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}_i(\tau_{<i}) \wedge e \end{array} \right\}\right| \\
&= \left|\left\{ \sigma \;\middle|\; \begin{array}{l} \tau = \sigma(\mathrm{tr}(P)) \\ \sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = s_{\mathcal{A}}(x_i)(\tau_{<i}) \wedge e \end{array} \right\}\right| \\
&= |\{\sigma \mid \sigma \models (\phi(P) \wedge e)[x_i \leftarrow s_{\mathcal{A}}(x_i)]\}| \\
&= \left|\left\{ \sigma\downharpoonright_{\mathbf{Rand}(P)} \;\middle|\; \sigma \models (\phi(P) \wedge e)[x_i \leftarrow s_{\mathcal{A}}(x_i)] \right\}\right| \\
&= \|\Phi^e_{\mathcal{R}}(P)(s_{\mathcal{A}})\|
\end{aligned}$$

□

Example 9.0.1 (continuing from p. 97). If we consider the quantitative reachability objective $x_4 \Leftrightarrow y_1$, the SSAT query is:

$$\underbrace{\text{Я}x_1.\ \mathsf{M}x_2.\ \text{Я}x_3.\ \mathsf{M}x_4.}_{\mathbf{Pref}(P)}\ \underbrace{\text{Я}y_1.\ \text{Я}y_2.}_{\mathbf{Rand}(P)}\ \underbrace{x_1 \Leftrightarrow (y_1 \vee y_2) \wedge x_3 \Leftrightarrow (x_2 \wedge y_2)}_{\phi(P)} \wedge x_4 \Leftrightarrow y_1$$

We partially unfold the assignment tree (only for variables in the prefix associated with P). This yields the following (with edges in red depicting the optimal strategy):


x_1
x_2
x_2
x_3
x_3
x_3
x_3
x_4
$\frac{1}{4}$ $\frac{1}{4}$ $\frac{1}{4}$ 0 0 0 $\frac{1}{2}$ $\frac{1}{4}$ 0 0 0 $\frac{1}{4}$ 0 0 0 $\frac{1}{4}$

This corresponds to the SSAT strategy:

$$s(x_2) = \top \qquad s(x_4) = x_1$$

Which is turns corresponds to the following attack:

$$\mathcal{A}_1(x_1) = \top \qquad \mathcal{A}_4(x_1, x_2, x_3) = x_1$$

9.4 Leakage

As we have defined all the necessary steps in the previous section, we can now define the equivalent of Theorem 9.3 with regard to leakage.

These two theorems only differ by the way we handle the variables in $\textbf{Rand}(P)$, counting them in one case and projecting them in the other. This is the interesting point in these reductions: one can encode these two seemingly different problems concisely, and the difference in the encoding is merely a difference in the prefix of the encoding.

Theorem 9.4 (Leakage to SSAT). *Given a* SIP *program P in normal form. The following are equivalent:*

- *The strategy $s_{\mathcal{A}^\star}$ is the optimal strategy for the SSAT query*

$$\textbf{Pref}(P).\ \mathsf{M}\textbf{Rand}(P).\ \mathsf{M}\textbf{Rest}(P).\ \phi(P)$$

- *The attack $\mathcal{A}^\star$, such that $\mathcal{A}^\star_i = s_{\mathcal{A}^\star}(x_i)$ is optimal with regard to leakage*

Proof. Here again, the proof follows closely the proof of the similar theorem in Chapter 6. Let $\mathcal{A}$ be an adaptive attack, i.e. such that $\mathcal{A}_i = s_{\mathcal{A}}(x_i)$, and let its associated SSAT strategy $s_{\mathcal{A}}$. Also, let $\Phi_{\mathcal{L}}(P)$ be the SSAT problem mentioned in the theorem.

It follows from Property 4.3 that it is sufficient to prove:

$$\left| [\![P]\!]_{\mathcal{A}} \downharpoonright_{\textbf{Trace}} \downharpoonright_{\text{out}} (\textbf{State}_0) \right| = \|\Phi_{\mathcal{L}}(P)(s_{\mathcal{A}})\|$$

The following is a consequence of Theorem 5.14:

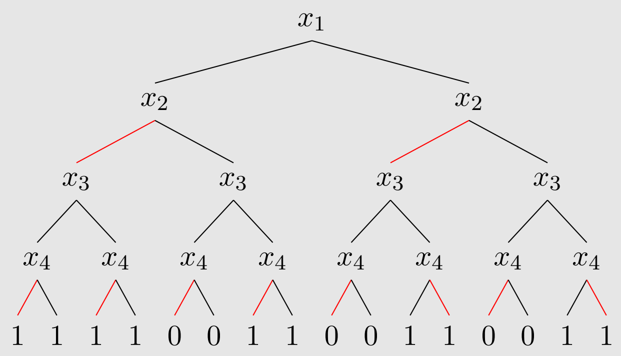
$$\begin{aligned}
\left|[\![P]\!]_{\mathcal{A}}\lfloor_{\mathbf{Trace}}\lfloor_{\text{out}}(\mathbf{State}_0)\right| &= |\{\tau\lfloor_{\text{out}} \mid \exists\sigma.\ (\sigma,\tau)\in[\![P]\!]_{\mathcal{A}}(\mathbf{State}_0)\}| \\
&= \left|\left\{\tau\lfloor_{\text{out}} \;\middle|\; \exists\sigma.\ \left|\begin{matrix}\tau=\sigma(\text{tr}(P)) \\ \sigma\models\phi(P)\wedge\bigwedge_{i=1,n} x_i=\mathcal{A}(\tau_{<i})\end{matrix}\right.\right\}\right| \\
&= \left|\left\{\sigma(\text{tr}(P))\lfloor_{\text{out}} \;\middle|\; \begin{matrix}\tau=\sigma(\text{tr}(P)) \\ \sigma\models\phi(P)\wedge\bigwedge_{i=1,n} x_i=\mathcal{A}_i(\tau_{<i})\end{matrix}\right\}\right| \\
&= \left|\left\{\sigma(\text{tr}(P))\lfloor_{\text{out}} \;\middle|\; \begin{matrix}\tau=\sigma(\text{tr}(P)) \\ \sigma\models\phi(P)\wedge\bigwedge_{i=1,n} x_i=s_{\mathcal{A}}(x_i)(\tau_{<i})\end{matrix}\right\}\right| \\
&= |\{\sigma(\text{tr}(P))\lfloor_{\text{out}} \mid \sigma\models\phi(P)[x_i\leftarrow s_{\mathcal{A}}(x_i)]\}| \\
&= \left|\left\{\sigma\lfloor_{\mathbf{Out}(P)} \;\middle|\; \sigma\models\phi(P)[x_i\leftarrow s_{\mathcal{A}}(x_i)]\right\}\right| \\
&= \|\Phi_{\mathcal{L}}(P)(s_{\mathcal{A}})\|
\end{aligned}$$

□

Example 9.0.1 (continuing from p. 97). The associated prefix stays the same as with the case of quantitative reachability. The SSAT query is thus:

$$\underbrace{Я x_1.\ \mathsf{M} x_2.\ Я x_3.\ \mathsf{M} x_4.}_{\mathbf{Pref}(P)}\ \underbrace{\mathsf{M} y_1.\ \mathsf{M} y_2.}_{\mathbf{Rand}(P)}\ \underbrace{x_1\Leftrightarrow(y_1\vee y_2)\wedge x_3\Leftrightarrow(x_2\wedge y_2)}_{\phi(P)}$$

Unfolding the assignment tree we get (with red edges depicting the optimal strategy):



We can see that the optimal strategy for leakage coincides with that of the robust reachability objective mentioned in Example 9.0.1.

Chapter 10

Experimental evaluation

In this chapter, we perform an experimental evaluation of the reduction we defined in Chapter 9. For that, we define some example programs, reduce them to normal form, and ask SSAT solvers to compute either the quantitative reachability, or the leakage of said program.

For all examples, we perform the normalization steps explained in Chapter 5 in order to get a boolean formula corresponding to the problem. Once that is done, we submit the instance to SSAT solvers to get an answer to the corresponding query. Note that not all SSAT solver are *witness-generating*: they do not return the strategy associated with the optimal answer, and they only evaluate the probability of the query[1].

Given that our programs act over the theory of bit-vectors, we can vary the size of the bit-vectors we consider, allowing to check the performances of the tools we use.

Table 10.1 shows details about the examples considered. The variable counts is provided over the bit-vector theory, and thus do not count the number of boolean variables after bit-blasting.

10.1 Quantitative reachability examples

We begin by defining multiple examples based on quantitative reachability.

Example 10.1.1. This is a slightly adapted version of Example 4.1.1 where we consider *wrapping* addition instead of saturating addition.

[1]They are thus SSAT solver and not F-SSAT solvers

Name	$\|\mathsf{M}(\Phi)\|$	$\|\text{Я}(\Phi)\|$
Example 4.1.1	1	2
Example 10.1.1	1	2
Example 10.1.2	3	4
Example 10.2.1	3	3

Table 10.1: Examples used in our benchmarks

```
x1 <- y1 + y2
out(x1)
x2 <- in()
```

The quantitative reachability objective of $y_1 \leq x_2 \leq y2$.

Example 10.1.2. The program is:

```
x1 <- in()
z1 <- x1 ≥ y1
out(z1)

x2 <- in()
z2 <- x2 ≥ y1
out(z2)

x3 <- in()
z3 <- x3 ≥ y1
out(z3)

x4 <- in()
```

The quantitative reachability objective is $x_4 = y_1$. The optimal strategy corresponds to dichotomic search among the possible values of y_1.

10.2 Leakage example

We also define a leakage based example. Note that this example (Example 10.2.1) is adapted from Example 10.1.2. The optimal strategies in both cases are the same, and the objective behind the similarity of this example is to check the impact of the encoding of the problem on solver performances.

Example 10.2.1. The objective is to maximize leakage:

```
x1 <- in()
z1 <- x1 ≥ y1
out(z1)

x2 <- in()
z2 <- x2 ≥ y1
out(z2)

x3 <- in()
z3 <- x3 ≥ y1
out(z3)
```

Example		`ClauSSat`			`SharpSSAT`	
		Time (in s)	Upper-bound	Lower-bound	Time (in s)	Result
4.1.1	4 bits	2.60	**0.028**	**0.028**	**0.25**	**0.028**
	8 bits	TO	0.89	1.238e-4	**9.62**	**0.0016**
10.1.1	4 bits	3.77	**0.024**	**0.024**	**0.25**	**0.024**
	8 bits	TO	0.67	4.662e-04	**4.83**	**0.0014**
10.1.2	4 bits	TO	1	0.0078	**0.575**	**0.0078**
	8 bits	TO	1	0	TO	-
10.2.1	4 bits	65.97	**0.5**	**0.5**	**0.12**	**0.5**
	8 bits	TO	1	0	TO	-

Table 10.2: Results of `ClauSSat` and `SharpSSAT` on our set of benchmarks

10.3 Tools used

To perform evaluation of the queries, we use `ClauSSat` [CHJ21] and `SharpSSAT` [FJ23].

`ClauSSat` algorithm leverages a technique called *clause selection* to speed up the evaluation of the probability of the formula. However, `ClauSSat` is not *witness-generating*, so the result returned by the tool will not allow us to generate the optimal strategy associated with the probability. `ClauSSat` accepts a time-out parameter. If the algorithm runs for more than the set amount of time, it will terminate and return bounds around the answer to the given SSAT query.

On the other hand `SharpSSAT` leverages the well-known technique of component caching from model counting to speed up the computation of the probability of the formula. This algorithm allows generating the strategy associated with the probability of the formula, allowing to synthesize the attack strategy for each program.

10.4 Results

We run both algorithms with their default parameters and a timeout of 1000 seconds on a machine with 2 Xeon Gold 6330 CPUs running a 2GHz, and 512GB of RAM. Results are shown in Table 10.2.

We can see that, as expected, the solving time of our instances grows with the number of bits. This is because the solver has the synthesize strategies for more bits, which augments the size of the prefix (but not the alternation in quantifiers).

The other comparison point concerns Examples 10.1.2 and 10.2.1. These two examples are two equivalent encodings of the same problem, which the optimal answer is dichotomic search. We can clearly see that encoding the problem as a leakage maximization problem is better than with quantitative reachability. This is because the quantitative reachability encoding adds one more quantifier level (because of the x4 `<- in()` statement at the end of Example 10.2.1).

This shows that while progress in solvers themselves can help, specific optimizations in terms of query encoding can drastically reduce the solving time.

In all cases, we can see that `SharpSSAT` is able to solve most examples in a very short time. It may be because our benchmarks are essentially *counting problems* rather than generic SSAT instances, and `SharpSSAT` exploits model-counting techniques to speed-up resolution.

Part V

Multiple attacks

Chapter 11

m-SIP: multi-channel SIP

We present in this chapter an extension of the SIP language. The idea is to account for a new class of security problems. This new class is concerned about security evaluation against *multiple attackers* with each their own knowledge of the current state of the program. Note that in this generalisation, the formal language is *strictly more expressive* through the use of channels (SIP only uses one). However the capacities of the individual attackers are lowered: they can only see what appears on their channels. Overall though, this increases the complexity of attack synthesis as one has to consider all possible combinations of all strategies of the attackers in order to compute the optimal overall strategy.

Intuitively, this is done by allowing multiple channels of communication, each being assigned a unique identifier and being controlled by a unique attacker. The common goal of the attackers will remain the same, i.e. maximizing either quantitative reachability, or global leakage (across all channels).

We will see in this chapter that the counting problem corresponding to this kind of attacks (i.e. multiple attack) is different and more computationally expensive. This is expected as this problem is indeed a strict generalization of the problems considered earlier in this thesis.

11.1 m-SIP syntax and semantics

We make the same assumptions as for the SIP language (see Section 4.1) but we further assume a certain number m of *channels*. Each of these channels are associated with a number in $\{1, \ldots, m\}$. We thus *generalize* input and output operations to specify the channel.

Example 11.1.1. The following is a valid m-SIP program:

```
x <- in(1)
out(x, 2)
y <- in(2)
if x == y then
  out(y, 1)
else
  skip
end
```

Channel	i	$\in$	$\{1, \ldots, m\}$	*channel identifier*
Expression	e	$::=$	x	*program variable in X*
		$\mid$	y	*random variable in Y*
		$\mid$	v	*value in* **Val**
		$\mid$	`ite`$(e$`,` e`,` $e)$	*if then else*
		$\mid$	$\star_1\ e$	*unary operation*
		$\mid$	$e\ \star_2\ e$	*binary operation*
Program	P	$::=$	x `<-` e	*assignment*
		$\mid$	x `<-` `in`(i)	*input*
		$\mid$	`out`$(x$`,` $i)$	*output*
		$\mid$	`skip`	*no operation*
		$\mid$	`if` e `then` P `else` P `end`	*conditional statement*
		$\mid$	P `;` P	*sequence*

Figure 11.1: m-SIP grammar

Note that the syntax of m-SIP (Figure 11.1) is very similar to that of SIP (Figure 4.1), the only difference being that input and output statement now accept a new parameter identifying the channel of communication. This simple change yields a slight adjustment in the semantics, which we will make explicit in the remainder of this section.

We only present the semantics of m-SIP under attack in this section. We now assume a set $\mathbb{A} \triangleq \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ of attacks, that will play *concurrently* against the program, without sharing knowledge. This means keep track of the current knowledge (i.e. the trace) separately for every channel.

There are two notable changes in Figure 11.2 compared to Figure 4.3. The first is about trace management: in the case of m-SIP, we compute an (indexed) set of traces, each corresponding to the communication on one of the channels. Second, the rules “Input” (Figure 11.2b) and “Output” (Figure 11.2a) change to account for the new structure to handle traces: one needs to update only the k-th trace. “Input” (Figure 11.2b) also changes the way the attack is invoked. Indeed, we now call the k-th attack with the τ_k (it's own trace) as an argument.

Note that we retain the determinism of SIP (Lemma 4.1). This will in turn ensure that we retain most of the properties of SIP.

Lemma 11.1. *Given an* m-SIP *program P and $(\sigma, \mathcal{T})$ a configuration, there exist a unique pair $(\sigma', \mathcal{T}')$ such that:*

$$(P, \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} * (\bullet, \sigma', \mathcal{T}')$$

Moreover:

$$\forall y \in \mathbf{Rand}(P).\ \sigma(y) = \sigma'(y)$$

Proof. Similar to Lemma 4.1 by definition of the semantics. □

Definition 11.1 (Final state)**.** Given an m-SIP program P and $(\sigma, \mathcal{T})$ a configuration, we define the *final state* of P when starting from configuration $(\sigma, \mathcal{T})$ as the unique pair $(\sigma', \mathcal{T}')$

$$\frac{\mathcal{T} = \{\tau_1, \dots, \tau_k, \dots, \tau_m\}}{(\texttt{out}(x,\ k), \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} (\bullet, \sigma, \mathcal{T}[\tau_k \leftarrow \tau_k \cdot \mathrm{out}_i(\sigma(x))])}$$

(a) Output

$$\frac{\mathbb{A} = \{\mathcal{A}_1, \dots, \mathcal{A}_k, \dots, \mathcal{A}_m\} \qquad \mathcal{T} = \{\tau_1, \dots, \tau_k, \dots, \tau_k\} \qquad v = \mathcal{A}_k(\tau_k)}{(x\ \texttt{<-}\ \texttt{in}(k), \sigma, \tau) \underset{\mathbb{A}}{\rightarrow} (\bullet, \sigma[x \leftarrow v], \mathcal{T}[\tau_k \leftarrow \tau_k \cdot \mathrm{in}_i(v)])}$$

(b) Input

$$\frac{\sigma(e) = \top}{(\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}, \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} (P_1, \sigma, \mathcal{T})} \qquad \frac{\sigma(e) = \bot}{(\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end}, \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} (P_2, \sigma, \mathcal{T})}$$

(c) Conditional statement

$$\frac{}{(x\ \texttt{<-}\ e, \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} (\bullet, \sigma[x \leftarrow \sigma(e)], \mathcal{T})}$$

(d) Assignment

$$\frac{}{(\texttt{skip}, \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} (\bullet, \sigma, \mathcal{T})}$$

(e) Skip

$$\frac{(P_1, \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} (P_1', \sigma', \mathcal{T}')}{(P_1\ \texttt{;}\ P_2, \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} (P_1'\ \texttt{;}\ P_2, \sigma', \mathcal{T}')} \qquad \frac{(P_1, \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} (\bullet, \sigma', \mathcal{T}')}{(P_1\ \texttt{;}\ P_2, \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} (P_2, \sigma', \mathcal{T}')}$$

(f) Sequence

Figure 11.2: m-SIP semantics under attack

such that:

$$(P, \sigma, \mathcal{T}) \underset{\mathbb{A}}{\rightarrow} * \ (\bullet, \sigma', \mathcal{T}')$$

We denote this pair as $[\![P]\!]_{\mathbb{A}}(\sigma, \mathcal{T}) \triangleq (\sigma', \mathcal{T}')$.

Now that we have defined final states in the case of m-SIP, it is easy to see that the definitions for quantitative reachability and leakage are the same as in the SIP case. Furthermore, because the execution is deterministic (Lemma 11.1),Properties 4.2 and 4.3 are preserved in this case too.

11.2 Normalizing m-SIP programs

Just like in Part II our objective will be to reduce m-SIP programs to formulas, such that a theorem similar to Theorem 5.14 holds.

Note that the only procedure that would change in the normalisation process is the linearisation, that is, getting rid of `if` statements. We thus define here all the necessary restriction for linearisation to apply. Note that the proofs of the results in this section are similar to that of Section 5.2.

Let us first define program equivalence in the context of m-SIP programs. The following definitions are the trivial lifting of the definitions for program equivalence from SIP to m-SIP.

Definition 11.2 (m-SIP configuration equality)**.** We say that two configurations $(\sigma_1, \mathcal{T}_1), (\sigma_2, \mathcal{T}_2)$ are *equal on* E whenever:

1. $\mathcal{T}_1 = \mathcal{T}_2$

2. $\forall v \in E, \sigma_1(x) = \sigma_2(x)$

We denote this equality as $(\sigma_1, \mathcal{T}_1) =_E (\sigma_2, \mathcal{T}_2)$.

Definition 11.3 (m-SIP program equivalence)**.** Given two m-SIP programs P_1 and P_2, and given $E \subseteq X \cup Y$, we say that P_1 and P_2 are equivalent with respect to E (denoted $P_1 \sim_E P_2$) whenever:

1. $\mathbf{Use}(P_1) \cup \mathbf{Use}(P_2) \subseteq E$

2. $\mathbf{Rand}(P_1) = \mathbf{Rand}(P_2)$

3. $\forall \mathbb{A}.\ \forall c_1, c_2.\ c_1 =_E c_2 \implies [\![P_1]\!]_{\mathbb{A}}(c_1) =_E [\![P_2]\!]_{\mathbb{A}}(c_2)$

11.2.1 Linearisation

In Section 5.2.1, we have seen a characterization of the SIP programs for which we have a linearisation procedure. The criterion is based on the shape of the trace: all executions of the program must exemplify the same sequence of inputs and outputs, regardless of the branch taken in `if` statements.

Since the model for the trace changes from SIP to m-SIP, the criterion for linearisation is also altered. Note that in SIP, alternation between inputs and outputs is *linear*: a given input depends on *all* the preceding outputs.

Conversely, in m-SIP the alternation between inputs and outputs defines a *partial order*: a given input depends on *some* preceding outputs (those on the corresponding channel). The type of a program thus needs to reflect that and is thus the partial order that it follows.

Example 11.2.1. Let the following m-SIP program P:

```
x1 <- in(1)
z1 <- x1 + 42
out(z1, 2)
x2 <- in(2)
```

In this example we can see two types of *dependencies*:

- A *data* dependency between the first two statements, as well as the middle two statements

- An *input-output* dependency between the last two statements

As exemplified in Example 11.2.1 we also need to take into account so-called *data* dependencies in m-SIP case. These dependencies are caused by data-flows from inputs on one channel to outputs on another channel. They force the order of evaluation in the program, and thus need to appear in the type system, as they influence the partial order corresponding to the program.

Definition 11.4. An m-SIP *base type* is an event of the form e_k^i where $e \in \{\text{in}, \text{out}\}$, representing the i-th event on channel k.

We denote as $\mathcal{E}_k$ the events on channel k, i.e. $\mathcal{E}_k \triangleq \{e_k^i \mid e \in \{\text{in}, \text{out}\}\}$. We also denote $\mathcal{E} \triangleq \bigcup_{k \in \{1,\ldots,m\}} \mathcal{E}_k$.

Using this definition of base types, we can define the partial order that the program induces on the events *accross* channels.

Definition 11.5. Let $T \subseteq \mathcal{E} \times \mathcal{E}$. T is an m-SIP *type* whenever:

1. T is a strict partial order
2. for any $k \in \{1, \ldots, m\}$, $T \cap (\mathcal{E}_k \times \mathcal{E}_k)$ is a strict total order

We call $T\lfloor_{\mathcal{E}_k} \triangleq T \cap (\mathcal{E}_k \times \mathcal{E}_k)$ the k-th *lane* of T.

Example 11.2.2. We represent m-SIP types graphically as graphs with events as vertices and edges representing precedence in the partial order. The following is a valid m-SIP type:

$$\begin{tikzcd}
\text{in}_1^1 \arrow[d] & \\
\text{out}_2^1 \arrow[r] & \text{in}_2^2
\end{tikzcd}$$

This is actually the type of the program in Example 11.2.1.

Informally, for a given type T, $T\lfloor_{\mathcal{E}_k}$ represents the *input-output* orders, as they relate events on the same channel with one another. All the other elements of T are *data* dependencies, and they are the only way to impose an order on events on different channels.

We now define two operations on types that are necessary for defining the type system for m-SIP programs. Definition 11.6 defines a *meet operation*, that will be useful to check that two branches in an m-SIP program have compatible types. Definition 11.7 defines a *concatenation* operation, that will always succeed, and that will allow to define the type rule for sequence operation.

Definition 11.6 (m-SIP type meet)**.** Given to m-SIP types T_1 and T_2, we define the *meet* $T_1 \sqcap T_2$ as the type T defined by the transitive closure of $(T_1 \cup T_2)$, whenever it is a valid m-SIP type.

Note that the meet operation might fail. Indeed, the union operation may create dependency loops between events, which in turns would not make a valid partial order.

Example 11.2.3. Let T_1 and T_2 be the following types:

$$\begin{tikzcd}
\mathrm{out}_1^1 \arrow[r] & \mathrm{in}_1^2 \\
\mathrm{in}_2^1 \arrow[u] \arrow[r] & \mathrm{in}_2^2
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathrm{out}_1^1 \arrow[r] & \mathrm{in}_1^2 \\
\mathrm{in}_2^1 \arrow[r] & \mathrm{in}_2^2
\end{tikzcd}$$

Then $T_1 \sqcap T_2 = T_1 \cup T_2 = T_1$.
Additionally, let T_3 and T_4 be the following types:

$$\begin{tikzcd}
\mathrm{out}_1^1 \arrow[r] & \mathrm{in}_1^2 \\
\mathrm{out}_2^1 \arrow[r] & \mathrm{in}_2^2 \arrow[ul]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathrm{out}_1^1 \arrow[r] & \mathrm{in}_1^2 \arrow[dl] \\
\mathrm{in}_2^1 \arrow[r] & \mathrm{in}_2^2
\end{tikzcd}$$

Then the meet operation fails, as the union of these types is the following, which is not a valid partial order:

$$\begin{tikzcd}
\mathrm{out}_1^1 \arrow[r] & \mathrm{in}_1^2 \arrow[dl] \\
\mathrm{in}_2^1 \arrow[r] & \mathrm{in}_2^2 \arrow[ul]
\end{tikzcd}$$

In order to define type concatenation, we assume that one can construct a variable dependency order for the program. This variable dependency order is the traditional *def-use* order, that induces the order of evaluation of statements in the program, and is a classical notion [ASU86]. We denote this *def-use* order on variables as $\prec_P$ for a given program P.

In order to apply this to events, we denote as $\mathrm{var}(e)$ the variable that corresponds to an event e.

Definition 11.7 (m-SIP type concatenation)**.** Given two m-SIP types T_1 and T_2, we define *concatenation* of the types with regard to P, denoted $T_1 \mathbin{;}_P T_2$ as the following:

- At the end of lane k of T_1, concatenate lane k of T_2 (renaming the events of T_2 as needed)
- If $\mathrm{var}\left(\mathrm{in}_k^i\right) \prec_P \mathrm{var}\left(\mathrm{out}_{k'}^{i'}\right)$ then add an edge between in_k^i and $\mathrm{out}_{k'}^{i'}$ to the newly formed type

Example 11.2.4. Let us consider the following program P_1:

```
x <- in(1)
y <- x + 3
out(y, 2)
z <- in(2)
```

The variable order is the following:

$$x \prec_P y$$

Furthermore we have:

$$\begin{aligned}\mathrm{var}\left(\mathrm{in}_1^1\right) &= x\\ \mathrm{var}\left(\mathrm{out}_2^1\right) &= y\\ \mathrm{var}\left(\mathrm{in}_2^2\right) &= z\end{aligned}$$

And hence the type of the whole program must include $\mathrm{in}_1^1 < \mathrm{out}_2^1$. Now looking at each channel of the program, we have the following types before concatenation of the first two lines with the last two:

$$\mathrm{in}_2^2 \qquad\qquad \begin{tikzcd} \mathrm{out}_2^1 \arrow[r] & \mathrm{in}_2^2 \end{tikzcd}$$

Putting all this together we obtain the final type:

$$\begin{tikzcd} \mathrm{in}_1^1 \arrow[d] & \\ \mathrm{out}_2^1 \arrow[r] & \mathrm{in}_2^2 \end{tikzcd}$$

Figure 11.3 exposes the type system to be used for m-SIP programs. Rules “Output” (Figure 11.3a) and “Input” (Figure 11.3b) are close to their SIP counterparts: we only create a graph with one node, representing the event. “Assignment” (Figure 11.3e) is the same as the SIP counterpart: we create an empty graph for these statements as they do not carry useful information for the traces on their own. “Conditional statement” (Figure 11.3c) makes direct use of Definition 11.6, this allows to check that the types on both branches of the `if` statements match and compute the type that captures the constraints on both sides. Finally, “Sequence” (Figure 11.3d) makes direct use of Definition 11.7, computing the dependencies on the whole program. This ensures that the order between the events on different channels is propagated, allowing to track *data* dependencies.

The goal of the type system is to capture the subset of programs for which we *are sure* there exists an equivalent linear program.

Remark 11.2.5. Note that when $m = 1$, m-SIP programs are essentially SIP programs. In this case, the type derivation of a given program follows the type derivation of the SIP type system.

Example 11.2.6. Let us consider the following program P:

$$\frac{}{\vdash \texttt{out}(x,\ k) : \mathrm{out}_k^1}$$

(a) Output

$$\frac{}{\vdash x \ \texttt{<-}\ \texttt{in}(k) : \mathrm{in}_k^1}$$

(b) Input

$$\frac{\vdash P_1 : T_1 \qquad \vdash P_2 : T_2 \qquad T' = T_1 \sqcap T_2}{\vdash \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \texttt{end} : T'}$$

(c) Conditional statement

$$\frac{\vdash P_1 : T_1 \qquad \vdash P_2 : T_2}{\vdash P_1\ \texttt{;}\ P_2 : T_1 \fatsemi_{P_1\ ;\ P_2} T_2}$$

(d) Sequence

$$\frac{}{\vdash x\ \texttt{<-}\ e : \epsilon}$$

(e) Assignment

Figure 11.3: Behavioral type system

```
if y1 then
  // Subprogram P1
  x <- in(1)
  out(x, 2)
  z <- in(2)
else
  // Subprogram P2
  out(y2, 2)
  x <- in(1)
  z <- in(2)
end
```

The respective types T_1 and T_2 of subprograms P_1 and P_2 are:

$$\begin{tikzcd}
\mathrm{in}_1^1 \arrow[d] & \\
\mathrm{out}_2^1 \arrow[r] & \mathrm{in}_2^2
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathrm{in}_1^1 & \\
\mathrm{out}_2^1 \arrow[r] & \mathrm{in}_2^2
\end{tikzcd}$$

The type of the whole program is thus the meet operation on T_1 and T_2, which yields:

$$\begin{tikzcd}
\mathrm{in}_1^1 \arrow[d] & \\
\mathrm{out}_2^1 \arrow[r] & \mathrm{in}_2^2
\end{tikzcd}$$

We claim that any well-typed program can actually be linearised. The proof is slightly more complicated than for SIP. Indeed, in this case one might need first to reorder the statements in the branches of `if` statements so that their orders exactly match across all channels. After this preprocessing step, one can use the rules defined in Figure 5.4 to perform linearisation, slightly changing the rules to account for channel identifiers.

Lemma 11.2. *Given an* m-SIP *program P such that $\vdash P : T$, there exists an equivalent program* **Lin**(P) *with no* `if` *statement.*

Proof. First, notice that we can define a subtype relation. Given types T_1 and T_2 we say that T_1 is a *supertype* of T_2 if $T_1 \subseteq T_2$. Informally a supertype is *less restrictive* in terms of order. We denote this relation as $T_1 \sqsubseteq T_2$. Furthermore, $T_1 \sqsubseteq T_1 \sqcap T_2$ and $T_2 \sqsubseteq T_1 \sqcap T_2$.

Second, recall that for any partial order T, one can build a total order $\overline{T}$ such that $T \sqsubseteq \overline{T}$.

Now, realize that for each program P such that $\vdash P : T$, one can permute the statements in the program, and build a new program $\overline{P}$ such that $\vdash \overline{P} : \overline{T}$ and $P \sim_{\mathbf{Vars}(P)} \overline{P}$. Indeed, the type system captures precisely the dependencies required between events, and thus the amount of freedom in the permutation. Furthermore, the meet operation (Definition 11.6) yields a supertype of both the types in each branch, and this ensures that $\overline{T}$ is also a supertype of each branch.

From here, one can apply the rules in Figure 5.4 and get a linearised program from $\overline{P}$. The proof of equivalence from $\overline{P}$ to $\mathbf{Lin}(\overline{P})$ is similar to that of Theorem 5.10. □

Example 11.2.6 (continuing from p. 117). Given the type of the whole program, note that the type of the program is already a total order. Permuting the statements in P to get $\overline{P}$, we get:

```
if y1 then
  x <- in(1)
  out(x, 2)
  z <- in(2)
else
  x <- in(1)
  out(y2, 2)
  z <- in(2)
end
```

From here, we can apply the rules in Figure 5.4, and we get the following linear program:

```
x <- in(1)
p <- ite(y1, x, y2)
out(p, 2)
z <- in(2)
```

11.2.2 Normal forms and characteristic formulas

Now that we have seen that m-SIP programs can be linearised, it is easy to realize that one can pursue the process of normalisation for SIP and adapt it to m-SIP.

We thus finish this section by setting up the definition necessary to lay out a claim similar to Theorem 5.14, relating the finals states of the program and boolean formulas and *symbolic traces*.

Definition 11.8. Given a linear m-SIP program P, we define $\mathrm{tr}(P)$ the symbolic trace of P inductively as follows:

$$\begin{aligned}
\mathrm{tr}(x \texttt{ <- in}(k)) &\triangleq [\mathrm{in}_k(x)] \\
\mathrm{tr}(\texttt{out}(x\texttt{, }k)) &\triangleq [\mathrm{out}_k(x)] \\
\mathrm{tr}(x \texttt{ <- } e) &\triangleq [] \\
\mathrm{tr}(P_1 \texttt{ ; } P_2) &\triangleq \mathrm{tr}(P_1) + \mathrm{tr}(P_2)
\end{aligned}$$

Example 11.2.7. Let us consider the following linear program P (which is adapted from Example 11.2.6):

```
x1 <- in(1)
x2 <- ite(y1, x1, y2)
out(x2, 2)
x3 <- in(2)
```

The symbolic trace of this program is:

$$[\mathrm{in}_1(x_1), \mathrm{out}_2(x_2), \mathrm{in}_2(x_3)]$$

We can now define normal program in the context of m-SIP. Actually, the definition is exactly the same, since the definition of symbolic trace is very similar to that of SIP.

Definition 11.9 (Normal m-SIP program)**.** An m-SIP program P is said to be in *normal form* if:

1. P is *linear* (i.e. it does not contain any `if` statement)
2. P is in SSA form
3. P acts over boolean variables only
4. The variables appearing in $\mathrm{tr}(P)$ are $(x_i)_{i\in\{1,\dots,n\}}$ and:
   - Any variable appears exactly once
   - They appear in the order $x_1, \dots, x_n$ within the symbolic trace

Given a variable $x_i \in \mathbf{In}(P)$ we denote as $m(x_i)$ its corresponding channel identifier.

Example 11.2.7 (continuing from p. 120). Program P in this example is in normal form. Furthermore, we have the following:

$$m(x_1) = 1 \qquad\qquad m(x_3) = 2$$

Remark 11.2.8. For the sake of simplicity, as the trace of m-SIP programs in normal forms are *linear*, one can still recover the trace for a given attack by projecting the program trace on one of the channels, which we denote as $\mathcal{T}\downharpoonleft_k$.

We define the characteristic formula $\phi(P)$ of a program P in normal form in the same fashion as in SIP. This is because the `in` and `out` statements have no contribution in the definition of the formula (they affect only quantifies in the prefix of the SSAT formulas).

Example 11.2.7 (continuing from p. 120). The characteristic formula of program P is:

$$x_2 = ite(y_1, x_1, y_2)$$

We can now state the final claim of this section, which is the m-SIP version of Theorem 5.14. The statement is slightly changed here, as one needs to account for the different channels of the program.

Theorem 11.3. *Given an* m-SIP *program P in normal form. The following are equivalent for every attack $\mathbb{A} = \{\mathcal{A}^1, \ldots, \mathcal{A}^m\}$ and configuration $(\sigma, \mathcal{T})$:*

1. $(\sigma, \mathcal{T}) \in [\![P]\!]_{\mathbb{A}}(\mathbf{State}_0)$
2. $\sigma \models \phi(P) \wedge \bigwedge_{x_i \in \mathbf{In}(P)} x_i = \mathcal{A}^{m(x_i)}\left(\mathcal{T}_{<i}\downharpoonleft_{m(x_i)}\right)$ *and* $\mathcal{T} = \sigma(\mathrm{tr}(P))$

Proof. The proof closely follows the one of Theorem 5.14, only changing steps to account for actions on the different channels. □

Chapter 12

DQMax#SAT

As with previous levels of the attacker hierarchy, we present here the counting problem corresponding to security objectives with regard to multiple attacks. Intuitively, this counting problem has to be a strict generalization of all the problems exposed before, given the new class of attacks it relates to.

This step up in generalization is done using so-called *Henkin quantifiers* [HK65]. Informally, these quantifiers allow specifying explicitly the dependencies between variables when quantifying in a formula. Contrary to *classical quantifiers* (i.e. $\forall$ and $\exists$ in first order logic) which occur sequentially within the prefix, Henkin quantifiers allow specifying a non-linear order for variable dependencies. This is of interest given the relation between SIP and m-SIP: the first needing linear dependencies between events, the second allowing strict partial orders in general.

12.1 Problem statement

Definition 12.1 (Syntax of DQMax#SAT)**.** Given $X = \{x_1, \ldots, x_n\}$, Y, and Z three sets of variables, a boolean formula $\phi(X, Y, Z)$ and $H_1, \ldots, H_n \subseteq Y \cup Z$, a DQMax#SAT query is of the form:

$$\Phi \triangleq \text{Я}Y.\ \exists Z.\ \mathsf{M}^{H_1}x_1.\ \ldots\ \mathsf{M}^{H_n}x_n.\ \phi$$

Each H_i is called the *dependency set* of x_i. We call *matrix* of Φ the formula $\mathbf{Mat}(\Phi) \triangleq \phi$. We also denote $\mathsf{M}(\Phi) \triangleq X$, $\text{Я}(\Phi) \triangleq Y$, and $\exists(\Phi) \triangleq Z$.

Example 12.1.1. Let Φ be the following DQMax#SAT query:

$$\text{Я}y_1.\ \text{Я}y_2.\ \exists z_1.\ \exists z_2.\ \mathsf{M}^{\{z_1\}}x_1.\ \mathsf{M}^{\{z_2\}}x_2.(x_1 \Rightarrow y_2) \wedge (y_1 \Rightarrow x_2) \wedge (y_1 \vee z_2 \Leftrightarrow y_2 \wedge z_1)$$

We have:

$$\mathsf{M}(\Phi) = \{x_1, x_2\} \qquad \text{Я}(\Phi) = \{y_1, y_2\} \qquad \exists(\Phi) = \{z_1, z_2\}$$

We now follows a process similar to Section 9.2, by first introducing the concept of *strategies* in the context of DQMax#SAT, and using this to define the corresponding counting problem.

Definition 12.2 (DQMax#SAT strategy)**.** Given a DQMax#SAT query Φ as in Definition 12.1, a *strategy* s is a replacement from variables in $\mathsf{M}(\Phi)$ to expressions over $Я(\Phi) \cup \exists(\Phi)$ respecting the dependencies. That is:

$$\forall x_i \in \mathsf{M}(\Phi).\ s(x_i) \in \mathcal{F}\langle H_i \rangle$$

We denote $\mathbf{Strats}(\Phi)$ the set of strategies for Φ.

Example 12.1.1 (continuing from p. 123)*.* $\mathbf{Strats}(\Phi)$ contains a total of 16 elements: there are only 4 possible replacements for x_1 (resp. x_2) depending only on the variable z_1 (resp z_2).

Following the definition of strategies, we define the count of a strategy, and finally define the DQMax#SAT problem, which is the counting problem that we will use to reduce security evaluation against multiple attacks.

Definition 12.3 (Count of a strategy)**.** Given a DQMax#SAT query Φ, we define the *count* of a strategy $s \in \mathbf{Strats}(\Phi)$ as:

$$\|\Phi(s)\| \triangleq \left|\left\{ y : Я(\Phi) \to \mathbb{B} \;\middle|\; \exists z : \exists(\Phi) \to \mathbb{B}.\ \mathbf{Mat}(\Phi)[x_i \leftarrow s(x_i)](y, z) \text{ is true} \right\}\right|$$

Definition 12.4 (DQMax#SAT problem)**.** Given a DQMax#SAT query Φ, the DQ-Max#SAT problem asks for finding a strategy $s^\star \in \mathbf{Strats}(\Phi)$ such that:

$$\|\Phi(s^\star)\| = \max_{s \in \mathbf{Strats}(\Phi)} \|\Phi(s)\|$$

Example 12.1.1 (continuing from p. 123)*.* An optimal solution for query Φ is the replacement $s = \{x_1 \mapsto \bot, x_2 \mapsto \overline{z_2}\}$. In this case we have $\|\Phi(s)\| = 3$.

We can see that this is a direct generalization of F-SSAT (and therefore of SSAT): it is enough to take $H_i = Я(\Phi)_{<i}$ to transform an SSAT query to a DQMax#SAT query.

12.2 Reducing to DQMax#SAT

In this section, we present theorems that relate attack synthesis on m-SIP programs to DQ-Max#SAT, linking strategies from the second to attacks from the first. As the proofs of these theorems follow closely the proofs of similar theorems in this thesis (Theorems 9.3 and 9.4), we omit them.

We begin by defining the DQMax#SAT prefix associated with a given m-SIP program in normal form. In the case of DQMax#SAT (compared to SSAT in Chapter 9), we only need to specify the dependency sets for each variable in $\mathbf{In}(P)$ for a given program P. Informally, the dependencies of an input variable are the values of the output variables appearing before it along the trace, and outputted on its channel.

Definition 12.5 (Dependencies of an input variable)**.** Given an m-SIP program P in normal form. We define the *dependencies* of a variable $x_i \in \mathbf{In}(P)$ as:

$$H(x_i) = \mathrm{tr}(P)_{<i}\lfloor_{m(x_i)}$$

Theorem 12.1 (Quantitative reachability to DQMax#SAT). *Given an* m-SIP *program P in normal form such that $\mathbf{In}(P) = \{x_1, \ldots, x_n\}$, and a quantitative reachability objective e. The following are equivalent:*

- *The strategy $s_{\mathbb{A}^\star}$ is the optimal strategy for the DQMax#SAT query*

$$Я\mathbf{Rand}(P).\ \mathsf{M}^{H(x_i)}x_i.\ \exists\mathbf{Out}(P).\ \exists\mathbf{Rest}(P).\ \phi(P) \land e$$

- *The attack $\mathbb{A}^\star = \{\mathcal{A}^1, \ldots, \mathcal{A}^m\}$, such that $s_{\mathbb{A}^\star}(x_i) = \mathcal{A}_i^{m(x_i)}$ is optimal with regard to the quantitative reachability objective e*

Example 12.2.1. Let P be the following program:

```
x1 <- in(1)
x2 <- x1 ∨ y1
out(x2, 2)
x3 <- in(2)
```

And let the quantitative reachability objective be:

$$e \triangleq x_3 \Leftrightarrow x_1$$

In this case, the corresponding DQMax#SAT query is:

$$Яy_1.\ \mathsf{M}^{\emptyset}x_1.\ \mathsf{M}^{\{x_2\}}x_3.\ \exists x_2.\ x_2 \Leftrightarrow (x_1 \lor y_2) \land x_3 \Leftrightarrow x_1$$

In this case, the optimal strategy is that both attack give the same constant answer, that is:

$$\mathcal{A}^1(\tau) = \top$$
$$\mathcal{A}^2(\tau) = \top$$

Using the attack defined as above, the quantitative reachability is 1.
For the same program, we choose the following attack objective:

$$e' \triangleq x_3 \Leftrightarrow y_1$$

The following attack is optimal:

$$\mathcal{A}^1(\tau) = \bot$$
$$\mathcal{A}^2([\text{out}(x_2)]) = x_2$$

The first attack sets up the state of the program such that $x_2 = y_1$. The second attack can then (implicitly) use this fact such that the quantitative reachability is also 1.

Theorem 12.2 (Leakage to DQMax#SAT). *Given an* m-SIP *program P in normal form such that $\mathbf{In}(P) = \{x_1, \ldots, x_n\}$. The following are equivalent:*

- *The strategy $s_{\mathbb{A}^\star}$ is the optimal strategy for the DQMax#SAT query*

$$Я\mathbf{Out}(P).\ \mathsf{M}^{H(x_i)} x_i.\ \exists \mathbf{Rand}(P).\ \exists \mathbf{Rest}(P).\ \phi(P)$$

- *The attack $\mathbb{A}^\star = \{\mathcal{A}^1, \ldots, \mathcal{A}^m\}$, such that $s_{\mathbb{A}^\star}(x_i) = \mathcal{A}_i^{m(x_i)}$ is optimal with regard to leakage*

Example 12.2.1 (continuing from p. 125). The DQMax#SAT query for leakage is:

$$Я x_2.\ \mathsf{M}^{\emptyset} x_1.\ \mathsf{M}^{\{x_2\}} x_3.\ \exists y_1.\ x_2 \Leftrightarrow (x_1 \vee y_2)$$

In this case the value of x_3 does not matter, and thus the optimal DQMax#SAT strategy is e.g.:

$$s(x_1) = \top \qquad s(x_3) = \bot$$

This translates to the following attack:

$$\mathcal{A}^1(\tau) = \top \qquad \mathcal{A}^2(\tau) = \bot$$

We have proven in this chapter that m-SIP is an extension of SIP to multiple attackers. We showed that performing quantitative security evaluation on this new formal language amounts to solving DQMax#SAT problems. We will now focus our interest on DQMax#SAT, proposing resolution methods and performing an experimental evaluation.

Chapter 13

Solving DQMax#SAT

We have shown in the previous chapters (Chapters 11 and 13) that we can reduce security evaluation against multiple attackers to DQMax#SAT. We thus focus our study on this problem in this chapter. We propose algorithms to solve the problem in practice by relying on existing tools and solvers.

Disclaimer The results exposed in this chapter are also exposed in a paper by the author [Vig+24].

13.1 Hardness of DQMax#SAT

We briefly discuss now the relationship between the DQMax#SAT problem and the Max#SAT, DQBF and DSSAT problems. It turns out that DQMax#SAT is at least as hard as all of them, as illustrated by the following reductions.

13.1.1 DQMax#SAT is at least as hard as Max#SAT

It is immediate to see that the Max#SAT problem is the particular case of the DQ-Max#SAT problem where there are no dependencies, that is, $H_1 = H_2 = ... = H_n = \emptyset$.

13.1.2 DQMax#SAT is at least as hard as DQBF

Let $X = \{x_1, ..., x_n\}$, Y be disjoint finite sets of Boolean variables and let $H_1, ..., H_n \subseteq Y$. Given a DQBF formula:

$$\forall Y.\ \exists^{H_1} x_1.\ ...\ \exists^{H_n} x_n.\ \Phi(X, Y) \tag{13.1}$$

The DQBF problem [PR79] asks to synthesize a substitution $\sigma_X^* : X \to \mathcal{F}\langle Y \rangle$ whenever one exists such that (i) $\sigma_X^*(x_i) \in \mathcal{F}\langle H_i \rangle$, for all $i \in [1, n]$ and (ii) $\Phi[\sigma_X^*]$ is valid. The DQBF problem is reduced to the DQMax#SAT problem:

$$\text{Я}Y.\ \mathsf{M}^{H_1} x_1.\ ...\ \mathsf{M}^{H_n} x_n.\ \Phi(X, Y) \tag{13.2}$$

By solving (13.2) one can solve the initial DQBF problem (13.1). Indeed, let $\sigma_X^* : X \to \mathcal{F}\langle Y \rangle$ be a solution for (13.2). Then, the DQBF problem admits a solution if and only if

$\|\Phi(\sigma_X^*)\| = 2^{|Y|}$. Moreover, σ_X^* is a solution for the problem (13.1) because (i) σ_X^* satisfies dependencies and (ii) $\Phi[\sigma_X^*]$ is valid as it belongs to $\mathcal{F}\langle Y\rangle$ and has $2^{|Y|}$ models. Note that through this reduction of DQBF to DQMax#SAT, the maximizing quantifiers in DQMax#SAT can be viewed as Henkin quantifiers [HK65] in DQBF with a quantitative flavor.

13.1.3 DQMax#SAT is at least as hard as DSSAT

Let $X = \{x_1, ..., x_n\}$, $Y = \{y_1, ..., y_m\}$ be disjoint finite sets of variables. A DSSAT formula is of the form:

$$Я^{p_1} y_1. \; ... \; Я^{p_m} \mathsf{M}^{H_1} x_1. \; ... \; \mathsf{M}^{H_n} x_n. \; y_m. \; \phi(X, Y) \tag{13.3}$$

where $p_1, ..., p_m \in [0, 1]$ are respectively the probabilities of variables $y_1, ..., y_m$ to be assigned $\top$ and $H_1, ..., H_n \subseteq Y$ are respectively the dependency sets of variables $x_1, ..., x_n$. Given a DSSAT formula (13.3), the probability of an assignment $\alpha_Y : Y \to \mathbb{B}$ is defined as

$$\mathbb{P}[\alpha_Y] \triangleq \prod_{i=1}^{m} \begin{cases} p_i & \text{if } \alpha_Y(y_i) = \top \\ 1 - p_i & \text{if } \alpha_Y(y_i) = \bot \end{cases}$$

This definition is lifted to formula $\Psi \in \mathcal{F}\langle Y\rangle$ by summing up the probabilities of its models, that is, $\mathbb{P}[\Psi] \triangleq \sum_{\alpha_Y \models \Psi} \mathbb{P}[\alpha_Y]$.

The DSSAT problem [LJ21] asks, for a given formula (13.3), to synthesize a substitution $\sigma_X^* : X \to \mathcal{F}\langle Y\rangle$ such that (i) $\sigma_X^*(x_i) \in \mathcal{F}\langle H_i\rangle$, for all $i \in [1, n]$ and (ii) $\mathbb{P}[\Phi[\sigma_X^*]]$ is maximal. If $p_1 = ... = p_m = \frac{1}{2}$ then for any substitution $\sigma_X : X \to \mathcal{F}\langle Y\rangle$ it holds $\mathbb{P}[\Phi[\sigma_X]] = \frac{\|\Phi(\sigma_X)\|}{2^m}$. In this case, it is immediate to see that solving (13.3) as a DQMax#SAT problem (i.e., by ignoring probabilities) would solve the original DSSAT problem. Otherwise, in the general case, one can use existing techniques such as [Cha+15] to transform arbitrary DSSAT problems (13.3) into equivalent ones where all probabilities are $\frac{1}{2}$ and solve them as above.

Note that while the reduction above from DSSAT to DQMax#SAT seems to indicate the two problems are rather similar, a reverse reduction from DQMax#SAT to DSSAT seems not possible in general. That is, recall that DQMax#SAT allows for a third category of *existential* variables Z which can occur in the dependencies sets H_i and which are not used for counting but are projected out. Yet, such problems arise naturally in our application domain as illustrated in Chapter 12. If no such existential variables exists or if they do not occur in the dependencies sets then one can a-priori project them from the objective Φ and syntactically reduce DQMax#SAT to DSSAT with $\frac{1}{2}$ probabilities on counting variables. However, projecting existential variables in a brute-force way may lead to an exponential blow-up of the objective formula Φ, an issue already explaining the hardness of projected model counting vs model counting [Azi+15; LM19]. Otherwise, in case of dependencies on existential variables, it is an open question if any direct reduction exists as these variables do not fit into the two categories of variables (counting, maximizing) occurring in DSSAT formulas.

From the complexity point of view, the decision version of DQMax#SAT can be shown to be NExpTime-complete and hence it lies in the same complexity class as DQBF [PRA01] and DSSAT [LJ21].

Theorem 13.1. *The decision version of DQMax#SAT is* NExpTime-*complete.*

Proof. Given a DQMax#SAT problem, notice that the solution corresponding to the optimal substitution for the *maximizing variables* can be guessed and constructed as a truth table in non-deterministic exponential time. Given the guessed substitution, the counting of the number of models projected on the *counting variables* and comparison against the threshold can be performed in exponential time, too. Overall, the whole procedure is done in nondeterministic exponential time, and hence DQMax#SAT belongs to the NExpTime complexity class.

Second, to see why DQMax#SAT is NExpTime-hard, one can check that the above reduction of the NExpTime-complete problem DQBF to DQMax#SAT can be done in polynomial time wrt the size of the initial problem. □ □

The following proposition provides an upper bound on the number of models corresponding to the solution of computable using projected model counting.

Proposition 13.2. *For any substitution $\sigma_X : X \to \mathcal{F}\langle Y \cup Z\rangle$ it holds:*

$$\|\Phi(\sigma_X)\| \leq \|\exists X.\ \Phi\|$$

13.2 Global Method

We show in this section that the DQMax#SAT problem can be directly reduced to a Max#SAT problem with an exponentially larger number of maximizing variables and exponentially bigger objective formula.

First, recall that any boolean formula $\varphi \in \mathcal{F}\langle H\rangle$ can be written as a finite disjunction of a subset M_φ of complete cubes from $\mathcal{M}\langle H\rangle$, that is, such that the following equivalences hold:

$$\varphi \iff \vee_{m\in M_\varphi} m \iff \vee_{m\in\mathcal{M}\langle H\rangle}(\llbracket m \in M_\varphi\rrbracket \wedge m)$$

Therefore, any formula $\varphi \in \mathcal{F}\langle H\rangle$ is uniquely *encoded* by the set of boolean values $\llbracket m \in M_\varphi\rrbracket$ denoting the membership of each complete monomial m to M_φ. We use this idea to encode the substitution of a maximizing variable x_i by some formula $\varphi_i \in \mathcal{F}\langle H_i\rangle$ by using a set of boolean variables $(x'_{i,m})_{m\in\mathcal{M}\langle H_i\rangle}$ denoting respectively $\llbracket m \in M_{\varphi_i}\rrbracket$ for all $m \in \mathcal{M}\langle H_i\rangle$. We now define the following Max#SAT problem:

$$(\mathsf{M}x'_{1,m}.\ )_{m\in\mathcal{M}\langle H_1\rangle} \dots (\mathsf{M}x'_{n,m}.\ )_{m\in\mathcal{M}\langle H_n\rangle} ЯY.\ \exists Z.\ \exists X.$$
$$\phi(X,Y,Z) \wedge \bigwedge_{i\in[1,n]} \big(x_i \Leftrightarrow \vee_{m\in\mathcal{M}\langle H_i\rangle}(x'_{i,m} \wedge m)\big) \quad (13.4)$$

The next theorem establishes the relation between the two problems.

Theorem 13.3. *$\sigma^*_X = \{x_i \mapsto \varphi^*_i\}_{i\in[1,n]}$ is a solution to the problem DQMax#SAT if and only if $\alpha^*_{X'} = \{x'_{i,m} \mapsto \llbracket m \in M_{\varphi^*_i}\rrbracket\}_{i\in[1,n],m\in\mathcal{M}\langle H_i\rangle}$ is a solution to Max#SAT problem* (13.4).

Proof. Let us denote

$$\phi'(X', X, Y, Z) \triangleq \phi(X, Y, Z) \wedge \bigwedge_{i\in[1,n]} \left(x_i \Leftrightarrow \vee_{m\in\mathcal{M}\langle H_i\rangle}(x'_{i,m} \wedge m)\right)$$

Actually, for any $\phi \in \mathcal{F}\langle X \cup Y \cup Z\rangle$ for any $\varphi_1 \in \mathcal{F}\langle H_1\rangle, ..., \varphi_n \in \mathcal{F}\langle H_n\rangle$ the following equivalence is valid:

$$\phi(X, Y, Z)[\{x_i \mapsto \varphi_i\}_{i\in[1,n]}] \Leftrightarrow \left(\exists X.\ \phi'(X', X, Y, Z)\right) \left[\{x'_{i,m} \mapsto [\![m \in M_{\varphi_i}]\!]\}_{i\in[1,n], m\in\mathcal{M}\langle H_i\rangle}\right]$$

Consequently, finding the substitution σ_X which maximize the number of Y-models of the left-hand side formula (that is, of $\exists Z.\ \phi(X, Y, Z)$) is actually the same as finding the valuation $\alpha_{X'}$ which maximizes the number of Y-models of the right-hand side formula (that is, $\exists Z.\ \exists X.\ \phi'(X', X, Y, Z)$). □

Example 13.2.1. Consider the following DQMax#SAT query Φ:

$$Я y_1.\ Я y_2.\ \exists z_1.\ \exists z_2.\ \mathsf{M}^{\{z_1,z_2\}} x_1.\ (x_1 \Leftrightarrow y_1) \wedge (z_1 \Leftrightarrow y_1 \vee y_2) \wedge (z_2 \Leftrightarrow y_1 \wedge y_2)$$

Let ϕ denote the matrix of this query. In this case, $\mathcal{F}\langle\{z_1, z_2\}\rangle = \{\top,\ \bot,\ z_1,\ \overline{z_1},\ z_2,\ \overline{z_2},\ z_1 \vee z_2,\ \overline{z_1} \vee z_2,\ z_1 \vee \overline{z_2},\ \overline{z_1} \vee \overline{z_2},\ z_1 \wedge z_2,\ \overline{z_1} \wedge z_2,\ z_1 \wedge \overline{z_2},\ \overline{z_1} \wedge \overline{z_2},\ z_1 \Leftrightarrow z_2,\ \overline{z_1 \Leftrightarrow z_2}\}$, and one shall consider every possible substitution. One can compute for instance $\phi[x_1 \mapsto \overline{z_1} \wedge \overline{z_2}] \equiv ((\overline{z_1} \wedge \overline{z_2}) \Leftrightarrow y_1) \wedge (z_1 \Leftrightarrow y_1 \vee y_2) \wedge (z_2 \Leftrightarrow y_1 \wedge y_2)$ which only has one model ($\{y_1 \mapsto \bot, y_2 \mapsto \top, z_1 \mapsto \top, z_2 \mapsto \bot\}$) and henceforth $\|\Phi(\{x_1 \mapsto \overline{z_1} \wedge \overline{z_2}\})\| = 1$. Overall, for this problem there exists four possible maximizing substitutions σ^* respectively $x_1 \mapsto z_1$, $x_1 \mapsto z_2$, $x_1 \mapsto z_1 \vee z_2$, $x_1 \mapsto z_1 \wedge z_2$ such that for all of them $\|\Phi(\sigma^*)\| = 3$.

The complete monomials over $\{z_1, z_2\}$ are:

$$\mathcal{M}\langle\{z_1, z_2\}\rangle = \{z_1 \wedge z_2, \overline{z_1} \wedge z_2, z_1 \wedge \overline{z_2}, \overline{z_1} \wedge \overline{z_2}\}$$

This problem is reduced to the following:

$$\begin{gathered}\mathsf{M}x'_{1,z_1z_2}.\ \mathsf{M}x'_{1,z_1\overline{z_2}}.\ \mathsf{M}x'_{1,\overline{z_1}z_2}.\ \mathsf{M}x'_{1,\overline{z_1}\overline{z_2}}.\ Я y_1.\ Я y_2.\ \exists z_1.\ \exists z_2.\ \exists x_1.\\ (x_1 \Leftrightarrow y_1) \wedge (z_1 \Leftrightarrow y_1 \vee y_2) \wedge (z_2 \Leftrightarrow y_1 \wedge y_2) \wedge\\ \left(x_1 \Leftrightarrow ((x'_{1,z_1z_2} \wedge z_1 \wedge z_2) \vee (x'_{1,z_1\overline{z_2}} \wedge z_1 \wedge \overline{z_2}) \vee (x'_{1,\overline{z_1}z_2} \wedge \overline{z_1} \wedge z_2) \vee (x'_{1,\overline{z_1}\overline{z_2}} \wedge \overline{z_1} \wedge \overline{z_2}))\right)\end{gathered}$$

A possible answer is $x'_{1,z_1z_2} \mapsto \top, x'_{1,z_1\overline{z_2}} \mapsto \top, x'_{1,\overline{z_1}z_2} \mapsto \bot, x'_{1,\overline{z_1}\overline{z_2}} \mapsto \bot$. This yields the solution $\sigma_X(x_1) = (z_1 \wedge z_2) \vee (z_1 \wedge \overline{z_2}) = z_1$ which is one of the optimal solutions.

13.3 Incremental Method

In this section we propose a first improvement with respect to the reduction in the previous section. It allows to control the blow-up of the objective formula in the reduced Max#SAT problem through an incremental process. Moreover, it allows in practice to find good approximate solutions early.

The incremental method consists in solving a sequence of related Max#SAT problems, each one obtained from the original DQMax#SAT problem and a reduced set of dependencies

$H'_1 \subseteq H_1$, ..., $H'_n \subseteq H_n$. Actually, if the sets of dependencies H'_1, ..., H'_n are chosen such that to augment progressively from $\emptyset$, ..., $\emptyset$ to H_1, ..., H_n by increasing only one of H'_i at every step then (i) it is possible to build every such MAX#SAT problem from the previous one by a simple syntactic transformation and (ii) most importantly, it is possible to steer the search for its solution knowing the solution of the previous one.

The incremental method relies therefore on an oracle procedure `max#sat` for solving MAX#SAT problems. We assume this procedure takes as inputs the sets X, Y, Z of maximizing, counting and existential variables, an objective formula $\phi \in \mathcal{F}\langle X \cup Y \cup Z\rangle$, an initial assignment $\alpha_0 : X \to \mathbb{B}$ and a filter formula $\psi \in \mathcal{F}\langle X\rangle$. The last two parameters are essentially used to restrict the search for maximizing solutions and must satisfy:

- $\psi[\alpha_0] = \top$, that is, the initial assignment α_0 is a model of Ψ and
- forall $\alpha : X \to \mathbb{B}$ if $\alpha \nvDash \psi$ then $\|\Phi(\alpha)\| \leq \|\Phi(\alpha_0)\|$, that is, any assignment α outside the filter ψ is at most as good as the assignment α_0.

Actually, whenever the conditions hold, the oracle can safely restrict the search for the optimal assignements within the models of Ψ. The oracle produces as output the optimal assignement $\alpha^* : X \to \mathbb{B}$ solving the MAX#SAT problem.

The incremental algorithm proposed in Algorithm 3 proceeds as follows:

- at lines 1-5 it prepares the arguments for the first call of the MAX#SAT oracle, that is, for solving the problem where $H'_1 = H'_2 = ... = H'_n = \emptyset$,
- at line 7 it calls to the MAX#SAT oracle,
- at lines 9-10 it chooses an index i_0 of some dependency set $H'_i \neq H_i$ and a variable $u \in H_{i_0} \setminus H'_{i_0}$ to be considered in addition for the next step, note that these steps open the door to variable selection heuristics (see [Kul21]),
- at lines 11-19 it prepares the argument for the next call of the MAX#SAT oracle, that is, it updates the set of maximizing variables X', it refines the objective formula Φ', it defines the new initial assignment α'_0 and the new filter Ψ' using the solution of the previous problem,
- at lines 6,20,22 it controls the main iteration, that is, keep going as long as sets H'_i are different from H_i,
- at line 23 it builds the expected solution, that is, convert the Boolean solution α'^* of the final MAX#SAT problem where $H'_i = H_i$ for all $i \in [1, n]$ to the corresponding substitution σ^*_X.

Finally, note that the application of substitution at line 15 can be done such that to preserve the CNF form of Φ'. That is, the application proceeds clause by clause, by using the following equivalences for every formula ψ and substitution $\sigma_{i_0,m,u} \triangleq \{x'_{i_0,m} \mapsto (x'_{i_0,mu} \wedge u) \vee (x'_{i_0,m\overline{u}} \wedge \overline{u})\}$:

$$(\psi \vee x'_{i_0,m})[\sigma_{i_0,m,u}] \Leftrightarrow (\psi \vee x'_{i_0,mu} \vee x'_{i_0,m\overline{u}}) \wedge (\psi \vee x'_{i_0,mu} \vee \overline{u}) \wedge (\psi \vee x'_{i_0,m\overline{u}} \vee u)$$
$$(\psi \vee \overline{x'_{i_0,m}})[\sigma_{i_0,m,u}] \Leftrightarrow (\psi \vee \overline{x'_{i_0,mu}} \vee \overline{u}) \wedge (\psi \vee \overline{x'_{i_0,m\overline{u}}} \vee u)$$

Algorithm 3 Incremental Algorithm

```
function Solve($X = \{x_1, \dots, x_n\}, Y, Z, H_1, \dots, H_n, \phi$)
    $H'_i \leftarrow \emptyset$ for all $i \in \{1, \dots, n\}$
    $X' \leftarrow \left\{x'_{i,\top} \,\middle|\, i \in \{1, \dots, n\}\right\}$
    $\phi' \leftarrow \phi \wedge \bigwedge_{i \in [1,n]} (x_i \Leftrightarrow x'_{i,\top})$
    $\alpha'_0 \leftarrow \{x'_{i,\top} \mapsto \bot\}_{i \in \{1,\dots,n\}}$
    $\psi' \leftarrow \top$
    repeat
        $\alpha'^* \leftarrow \texttt{max\#sat}(X', Y, Z \cup X, \phi', \alpha'_0, \psi')$
        if $H'_i \neq H_i$ for some $i \in \{1, \dots, n\}$ then
            $i_0 \leftarrow \texttt{choose}(\{i \in [1,n] \mid H'_i \neq H_i\})$
            $u \leftarrow \texttt{choose}(H_{i_0} \setminus H'_{i_0})$
            $\alpha'_0 \leftarrow \alpha'^*$
            $\Psi' \leftarrow \bot$
            for all $m \in \mathcal{M}\langle H'_{i_0} \rangle$ do
                $X' \leftarrow (X' \setminus \{x'_{i_0,m}\}) \cup \{x'_{i_0,mu}, x'_{i_0,m\overline{u}}\}$
                $\phi' \leftarrow \phi'[x'_{i_0,m} \leftarrow (x'_{i_0,mu} \wedge u) \vee (x'_{i_0,m\overline{u}} \wedge \overline{u})]$
                $\alpha'_0 \leftarrow (\alpha'_0 \setminus \{x'_{i_0,m} \mapsto \_\}) \cup \{x'_{i_0,mu}, x'_{i0,m\overline{u}} \mapsto \alpha'_0(x'_{i_0,m})\}$
                $\psi' \leftarrow \psi' \vee (x'_{i_0,mu} \not\Leftrightarrow x'_{i_0,m\overline{u}})$
            end for
            $\psi' \leftarrow \psi' \vee \bigwedge_{x \in X'} (x \Leftrightarrow \alpha'_0(x))$
            $H'_{i_0} \leftarrow H'_{i_0} \cup \{u\}$
        end if
    until $H'_i = H_i$ for all $i \in \{1, \dots, n\}$
    $\sigma^*_X \leftarrow \{x_i \mapsto \vee_{m \in \mathcal{M}\langle H_i \rangle} (\alpha'^*(x'_{i,m}) \wedge m)\}_{i \in \{1,\dots,n\}}$
end function
```

Theorem 13.4. *Algorithm 3 is correct for solving the DQMax#SAT problem.*

Proof. The algorithm terminates after $1 + \sum_{i \in [1,n]} |H_i|$ oracle calls. Moreover, every oracle call solves correctly the Max#SAT problem corresponding to DQMax#SAT problem

$$\mathsf{M}^{H'_1} x_1. \; \dots \mathsf{M}^{H'_n} x_n. \; Я Y. \; \exists Z. \; \phi(X, Y, Z)$$

This is an invariance property provable by induction. It holds by construction of X', ϕ', α'_0, Ψ' at the initial step. Then, it is preserved from one oracle call to the next one i.e., X' and Φ' are changed such that to reflect the addition of the variable u of the set H'_{i_0}. The new initial assignment α'_0 is obtained (i) by replicating the optimal value $\alpha'^*(x'_{i_0,m})$ to the newly introduced $x'_{i_0,mu}, x'_{i_0,m\overline{u}}$ variables derived from $x'_{,m}$ variable (line 16) and (ii) by keeping the optimal value $\alpha'^*(x'_{i,m})$ for other variables (line 11). As such, for the new problem, the assignment α'_0 has exactly the same number of Y-projected models as the optimal assignment α'^* had on the previous problem. The filter ψ' is built such that to contain this new initial assignment α'_0 (line 19) as well as any other assignment that satisfies $x'_{i_0,mu} \not\Leftrightarrow x'_{i_0,m\overline{u}}$ for some monomial m (lines 12, 17). This construction guarantees that, any assignment which does not satisfy the filter Ψ' reduces precisely to an assignment of the previous problem, other than the optimal one α'^{*}, and henceforth at most as good as α'_0 regarding the number of Y-projected models. Therefore, it is a sound filter and can be used to restrict the search for the new problem. The final oracle call corresponds to solving the complete Max#SAT problem (13.4) and it will therefore allow to derive a correct solution to the initial DQMax#SAT problem. □

Example 13.3.1. Let reconsider Example 13.2.1. The incremental algorithm will perform 3 calls to the Max#SAT oracle. The first call corresponds to the problem

$$\mathsf{M}x'_{1,\top}.\ Яy_1.\ Яy_2.\ \exists z_1.\ \exists z_2.\ \exists x_1.\ (x_1 \Leftrightarrow y_1) \wedge (z_1 \Leftrightarrow y_1 \vee y_2) \wedge (z_2 \Leftrightarrow y_1 \wedge y_2) \wedge (x_1 \Leftrightarrow x'_{1,\top})$$

A solution found by the oracle is e.g., $x'_{1,\top} \mapsto \bot$ which has 2 projected models. If z_1 is added to H'_1, the second call corresponds to the refined Max#SAT problem:

$$\begin{aligned}&\mathsf{M}x'_{1,z_1}.\ \mathsf{M}x'_{1,\overline{z_1}}\ Яy_1.\ Яy_2.\ \exists z_1.\ \exists z_2.\ \exists x_1.\\ &\qquad (x_1 \Leftrightarrow y_1) \wedge (z_1 \Leftrightarrow y_1 \vee y_2) \wedge (z_2 \Leftrightarrow y_1 \wedge y_2) \wedge (x_1 \Leftrightarrow x'_{1,z_1} \wedge z_1 \vee x'_{1,\overline{z_1}} \wedge \overline{z_1})\end{aligned}$$

A solution found by the oracle is e.g., $x'_{1,z_1} \mapsto \top, x'_{1,\overline{z_1}} \mapsto \bot$ which has 3 projected models. Finally, z_2 is added to H'_1 therefore the third call corresponds to the complete Max#SAT problem as presented in Example 13.2.1. The solution found by the oracle is the same as in Example 13.2.1.

A first benefit of Algorithm 3 is the fact that it opens the door to any-time approaches to solve the DQMax#SAT problem. Indeed, the distance between the current and the optimal solution (that is, the relative ratio between the corresponding number of Y-projected models) can be estimated using the upper bound provided by Prop. 13.2. Hence, one could stop the search at any given iteration as soon as some threshold is reached and construct the returned value σ_X similarly as in Line 24 in Algorithm 3. In this case the returned σ_X would be defined as $\sigma_X = \{x_i \mapsto \vee_{m \in \mathcal{M}\langle H'_i\rangle} (\alpha'^{*}(x'_{i,m}) \wedge m)\}_{i\in[1,n]}$ (note here that the monomials are selected from H'_i instead of H_i).

Another benefit of the incremental approach is that it is applicable without any assumptions on the underlying Max#SAT solver. Indeed, one can use ψ' in Algorithm 3 by solving the Max#SAT problem corresponding to $\phi' \wedge \psi'$, and return the found solution. Even though the α'_0 parameter requires an adaptation of the Max#SAT solver in order to ease the search of a solution, one could still benefit from the incremental resolution of DQMax#SAT. Notice that a special handling of the ψ' parameter by the solver would avoid complexifying the formula passed to the Max#SAT solver and still steer the search properly.

13.4 Local Method

The local resolution method allows to compute the solution of an initial DQMax#SAT problem by combining the solutions of two strictly smaller and independent DQMax#SAT sub-problems derived syntactically from the initial one. The local method applies only if either 1) some counting or existential variable u is occurring in all dependency sets; or 2) if there is some maximizing variable having an empty dependency set. That is, in contrast to the global and incremental methods, the local method is applicable only in specific situations.

Given a DQMax#SAT problem of form and a variable v, let $\phi_v \triangleq \phi[v \leftarrow \top]$, $\phi_{\overline{v}} \triangleq \phi[v \leftarrow \bot]$ be the two cofactors on variable v of the objective ϕ.

13.4.1 Reducing Common Dependencies

Let us consider now a variable u which occurs in all dependency sets H_i and let us consider the following u-reduced DQMax#SAT problems:

$$\text{Я}\, Y \setminus \{u\}.\ \exists Z \setminus \{u\}.\ \mathsf{M}^{H_1\setminus\{u\}} x_1.\ \ldots \mathsf{M}^{H_n\setminus\{u\}} x_n.\ \phi_u \tag{13.5}$$
$$\text{Я}\, Y \setminus \{u\}.\ \exists Z \setminus \{u\}.\ \mathsf{M}^{H_1\setminus\{u\}} x_1.\ \ldots \mathsf{M}^{H_n\setminus\{u\}} x_n.\ \phi_{\overline{u}} \tag{13.6}$$

Let $\sigma^*_{X,u}$, $\sigma^*_{X,\overline{u}}$ denote respectively the solutions to the problems above.

Theorem 13.5. *If either*

(i) $u \in Y$ *or*

(ii) $u \in Z$ *and u is functionally dependent on counting variables Y within the objective ϕ (that is, for any valuation $\alpha_Y : Y \to \mathbb{B}$, at most one of $\phi[\alpha_Y][u \leftarrow \top]$ and $\phi[\alpha_Y][u \leftarrow \bot]$ is satisfiable).*

then σ^*_X defined as

$$\sigma^*_X(x_i) \triangleq \Big(u \wedge \sigma^*_{X,u}(x_i)\Big) \vee \Big(\overline{u} \wedge \sigma^*_{X,\overline{u}}(x_i)\Big) \text{ for all } i \in [1,n]$$

is a solution to the DQMax#SAT problem.

Proof. First, any formula $\varphi_i \in \mathcal{F}\langle H_i\rangle$ can be equivalently written as $u \wedge \varphi_{i,u} \vee \overline{u} \wedge \varphi_{i,\overline{u}}$ where $\varphi_{i,u} \triangleq \varphi_i[u \leftarrow \top] \in \mathcal{F}\langle H_i \setminus \{u\}\rangle$ and $\varphi_{i,\overline{u}} \triangleq \varphi_i[u \leftarrow \bot] \in \mathcal{F}\langle H_i \setminus \{u\}\rangle$. Second, we can prove the equivalence:

$$\begin{aligned}\phi[x_i \leftarrow \varphi_i] &\Leftrightarrow ((u \wedge \phi_u) \vee (\overline{u} \wedge \phi_{\overline{u}}))[x_i \leftarrow u \wedge \varphi_{i,u} \vee \overline{u} \wedge \varphi_{i,\overline{u}}] \\ &\Leftrightarrow (u \wedge \phi_u[x_i \leftarrow \varphi_{i,u}]) \vee (\overline{u} \wedge \phi_{\overline{u}}[x_i \leftarrow \varphi_{i,\overline{u}}])\end{aligned}$$

by considering the decomposition of Φ_u, $\Phi_{\overline{u}}$ according to the variable x_i.

The equivalence above can then be generalized to a complete substitution $\sigma_X = \{x_i \mapsto \varphi_i\}_{i\in[1,n]}$ of maximizing variables. Let us denote respectively $\sigma_{X,u} \triangleq \{x_i \mapsto \varphi_{i,u}\}_{i\in[1,n]}$, $\sigma_{X,\overline{u}} \triangleq \{x_i \mapsto \varphi_{i,\overline{u}}\}_{i\in[1,n]}$. Therefore, one obtains

$$\begin{aligned}\phi[\sigma_X] &\Leftrightarrow (u \wedge \phi_u \vee \overline{u} \wedge \phi_{\overline{u}})[x_i \leftarrow \varphi_i]_{i\in[1,n]} \\ &\Leftrightarrow \big(u \wedge \phi_u[x_i \leftarrow \varphi_{i,u}]_{i\in[1,n]}\big) \vee \big(\overline{u} \wedge \phi_{\overline{u}}[x_i \leftarrow \varphi_{i,\overline{u}}]_{i\in[1,n]}\big) \\ &\Leftrightarrow (u \wedge \phi_u[\sigma_{X,u}]) \vee (\overline{u} \wedge \phi_{\overline{u}}[\sigma_{X,\overline{u}}])\end{aligned}$$

Third, the later equivalence provides a way to compute the number of Y-models of the formula $\exists Z.\ \phi[\sigma_Z]$ as follows:

$$\begin{aligned}\|\Phi(\sigma_x)\| &= \|(u \wedge \Phi_u(\sigma_{X,u})) \vee (\overline{u} \wedge \Phi_{\overline{u}}(\sigma_{X,\overline{u}}))\| \\ &= \|(u \wedge \Phi_u)(\sigma_{X,u})\| + \|(\overline{u} \wedge \Phi_{\overline{u}})(\sigma_{X,\overline{u}})\|\end{aligned}$$

Note that the last equality holds only because $u \in Y$ or $u \in Z$ and functionally dependent on counting variables Y. Actually, in these situations, the sets of Y-projected models of respectively, $u \wedge \Phi_u[\sigma_{X,u}]$ and $\overline{u} \wedge \Phi_{\overline{u}}[\sigma_{X,\overline{u}}]$ are disjoint. Finally, finding σ_X which maximizes the left hand side reduces to finding $\sigma_{X,u}$, $\sigma_{X,\overline{u}}$ which maximizes independently the two terms of right hand side, and these actually are the solutions of the two u-reduced problems (13.5) and (13.6). □

Example 13.2.1 (continuing from p. 130). It is an immediate observation that existential variables z_1, z_2 are functionally dependent on counting variables y_1, y_2 according to the objective. Therefore the local method is applicable and henceforth since $H_1 = \{z_1, z_2\}$ one reduces the initial problem to four smaller problems, one for each valuation of z_1, z_2, as follows:

$$\begin{aligned}
z_1, z_2 \mapsto \top, \top &\quad : \quad \max^{\emptyset} x_1.\ Яy_1.\ Яy_2\ .(x_1 \Leftrightarrow y_1) \wedge (\top \Leftrightarrow y_1 \vee y_2) \wedge (\top \Leftrightarrow y_1 \wedge y_2) \\
z_1, z_2 \mapsto \top, \bot &\quad : \quad \max^{\emptyset} x_1.\ Яy_1.\ Яy_2\ .(x_1 \Leftrightarrow y_1) \wedge (\top \Leftrightarrow y_1 \vee y_2) \wedge (\bot \Leftrightarrow y_1 \wedge y_2) \\
z_1, z_2 \mapsto \bot, \top &\quad : \quad \max^{\emptyset} x_1.\ Яy_1.\ Яy_2\ .(x_1 \Leftrightarrow y_1) \wedge (\bot \Leftrightarrow y_1 \vee y_2) \wedge (\top \Leftrightarrow y_1 \wedge y_2) \\
z_1, z_2 \mapsto \bot, \bot &\quad : \quad \max^{\emptyset} x_1.\ Яy_1.\ Яy_2\ .(x_1 \Leftrightarrow y_1) \wedge (\bot \Leftrightarrow y_1 \vee y_2) \wedge (\bot \Leftrightarrow y_1 \wedge y_2)
\end{aligned}$$

The four problems are solved independently and have solutions e.g., respectively $x_1 \mapsto c_1 \in \{\top\}$, $x_1 \mapsto c_2 \in \{\top, \bot\}$, $x_1 \mapsto c_3 \in \{\top, \bot\}$, $x_1 \mapsto c_4 \in \{\bot\}$. By recombining these solutions according to Theorem 13.5 one obtains several solutions to the original DQMax#SAT problem of the form:

$$x_1 \mapsto (z_1 \wedge z_2 \wedge c_1) \vee (z_1 \wedge \overline{z_2} \wedge c_2) \vee (\overline{z_1} \wedge z_2 \wedge c_3) \vee (\overline{z_1} \wedge \overline{z_2} \wedge c_4)$$

They correspond to solutions already presented in Page 130, that is:

$$\begin{aligned}
x_1 &\mapsto (z_1 \wedge z_2 \wedge \top) \vee (z_1 \wedge \overline{z_2} \wedge \bot) \vee (\overline{z_1} \wedge z_2 \wedge \bot) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \bot) && (\equiv z_1 \wedge z_2) \\
x_1 &\mapsto (z_1 \wedge z_2 \wedge \top) \vee (z_1 \wedge \overline{z_2} \wedge \bot) \vee (\overline{z_1} \wedge z_2 \wedge \top) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \bot) && (\equiv z_2) \\
x_1 &\mapsto (z_1 \wedge z_2 \wedge \top) \vee (z_1 \wedge \overline{z_2} \wedge \top) \vee (\overline{z_1} \wedge z_2 \wedge \bot) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \bot) && (\equiv z_1) \\
x_1 &\mapsto (z_1 \wedge z_2 \wedge \top) \vee (z_1 \wedge \overline{z_2} \wedge \top) \vee (\overline{z_1} \wedge z_2 \wedge \top) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \bot) && (\equiv z_1 \vee z_2)
\end{aligned}$$

Finally, note that the local resolution method has potential for parallelization. It is possible to eliminate not only one but all common variables in the dependency sets as long as they fulfill the required property. This leads to several strictly smaller sub-problems that can be solved in parallel. The situation has been already illustrated in the previous example, where by the elimination of z_1 and z_2 one obtains 4 smaller sub-problems.

13.4.2 Solving Variables with no Dependencies

Let us consider now a maximizing variable which has an empty dependency set. Without lack of generality, assume x_1 has an empty dependency set, i.e. $H_1 = \emptyset$. Thus, the only possible values that can be assigned to x_1 are $\top$ or $\bot$. Let us consider the following x_1-reduced DQMAX#SAT problems:

$$\mathsf{M}^{H_2} x_2. \; \ldots \mathsf{M}^{H_n} x_n. \; Я\, Y. \; \exists\, Z. \; \phi_{x_1}$$
$$\mathsf{M}^{H_2} x_2. \; \ldots \mathsf{M}^{H_n} x_n. \; Я\, Y. \; \exists\, Z. \; \phi_{\overline{x_1}}$$

and let σ^*_{X,x_1}, $\sigma^*_{X,\overline{x_1}}$ denote respectively the solutions to the problems above. The following proposition is easy to prove, and provides the solution of the original problem based on the solutions of the two smaller sub-problems.

Proposition 13.6. *The substitution σ^*_X defined as follows is a solution to the DQMAX#SAT problem:*

$$\sigma^*_X \triangleq \begin{cases} \sigma^*_{X,x_1} \uplus \{x_1 \mapsto \top\} & \textit{if } \left\| \Phi_{x_1}\left(\sigma^*_{X,x_1}\right) \right\| \geq \left\| \Phi_{\overline{x_1}}\left(\sigma^*_{X,\overline{x_1}}\right) \right\| \\ \sigma^*_{X,\overline{x_1}} \uplus \{x_1 \mapsto \bot\} & \textit{otherwise} \end{cases}$$

Chapter 14

Experimental evaluation

In our implementation of Algorithm 3, we leave generic the choice of the underlying MAX#SAT solver. For concrete experiments, we used both the approximate solver `BaxMC`[1] [Vig+22] and the exact solver `D4max` [ALM22].

In the implementation of Algorithm 3 in our tool, the filter ψ' is handled as discussed at the end of Section 13.3: the formula effectively solved is $\phi' \wedge \psi'$, allowing to use any MAX#SAT solver without any prior modification. Remark that none of `BaxMC` and `D4max` originally supported exploiting the α_0 parameter of Algorithm 3 out of the box. While `D4max` is used of the shelf, we modified `BaxMC` to actually support this parameter for the purpose of the experiment.

We use the various examples used in Chapters 11 and 12 as benchmark instances for the implemented tool. Examples 12.1.1 and 13.2.1 are used as they are.

We add Examples 10.1.1 and 10.2.1, based on adaptive attack synthesis. As discussed in Chapter 12, these programs are bit-blasted and turned into DQMAX#SAT queries. We give them as programs for the sake of simplicity. Furthermore, the examples correspond to adaptive attack synthesis, thus the number of the channel is omitted in the examples for simplicity.

We also add the following security related problem (which corresponds to Program 5 from [Vig+23]) into our benchmark set:

Example 14.0.1.

$$\text{Я} y_1.\ \exists z_1.\ \exists z_2.\ \mathsf{M}^{\emptyset} x_1.\ \mathsf{M}^{\{z_1\}} x_2.\ \mathsf{M}^{\{z_1,z_2\}} x_3.\ (x_3 = y_1) \wedge (z_1 = (x_1 \geq y_1) \wedge z_2 = (x_2 \geq y))$$

When bit-blasting is needed for a given benchmark, the number of bits used for bit-blasting is indicated in parentheses. After the bit-blasting operation, the problems can be considered medium sized. Bigger bit-vector sizes resulted in instance where none of the solver could give an answer in the allowed time/memory limit, and are thus excluded from the table.

As you can see in Table 14.1, the implemented tool can effectively solve all the examples presented in this paper. We run both algorithms with their default parameters and a timeout of 30 minutes on a machine with 2 Xeon Gold 6330 CPUs running a 2GHz, and 512GB of

[1]Thanks to specific parametrization and the oracles [CMV13] used internally by `BaxMC`, it can be considered an exact solver on the small instances of interest in this section.

Table 14.1: Summary of the performances of the tool. $|\Phi|$ denotes the number of clauses. The last two columns indicate the running time using the specific Max#SAT oracle.

Benchmark name	$\|X\|$	$\|Y\|$	$\|Z\|$	$\|\Phi\|$	Max Models	Time (`BaxMC`)	Time (`D4max`)
Example 12.1.1	1	2	2	7	3	32ms	121ms
Example 13.2.1	2	2	2	7	3	25ms	134ms
Example 10.1.1 (3 bits)	3	6	97	329	6	378ms	79.88s
Example 10.1.1 (4 bits)	4	8	108	385	28	638.63s	$>$ 30mins
Example 14.0.1 (3 bits)	9	3	93	289	4	74.00s	18.62s
Example 10.2.1 (3 bits)	9	3	114	355	8	9.16s	93.48s

RAM. The synthesized answers (i.e. the monomials selected in Algorithm 3, Line 24) returned by both oracles are the same, that is, `BaxMC` returned an optimal answer in all cases despite being theoretically probably approximate.

For security examples, one key part of the process is the translation of the synthesized answer (over boolean variables) back to the original problem (over bit-vectors). In order to do that, one can simply concatenate the generated sub-functions for each bit of the bit-vector into a complete formula, but that would lack explainability because the thus-generated function would be a concatenation of potentially big sums of monomials. In order to ease visual inspection, we run a generic simplification step [GMK15] for all the synthesized sub-function, before concatenation. This simplification allows us to directly derive the answers explicited in Examples 10.1.1 and 10.2.1 instead of their equivalent formulated as sums of monomials, and better explain the results returned by the tool.

Unfortunately, we could not compare our algorithm against the state-of-the-art DSSAT solver `DSSATpre` [LJ21] on the set of examples described in this paper because

1. as discussed in Section 13.1.3, some DQMax#SAT instances cannot be converted into DSSAT instances,

2. for the only DQMax#SAT instance (Example 10.2.1) that can be converted into a DSSAT instance, we were not able to get an answer using `DSSATpre`

.

Part VI

Conclusion

Chapter 15

Conclusion and perspectives

15.1 Conclusion

Throughout this thesis, we developed a method to evaluate quantitatively the security of a program. This new framework asks for the synthesis of an optimal attack with regard to a fixed objective, unifying and extending both *quantitative reachability* [BG22] and *leakage* [Smi09].

This unified framework also allows different *kinds* of attackers organized in a hierarchy and that we call the *attacker hierarchy*: *non-adaptive*, *adaptive*, and *multiple adaptive*. We showed that synthesis of the optimal attack (at each level of the hierarchy) corresponds to a specific *counting problem*: respectively MAX#SAT, F-SSAT and DQMAX#SAT. Reduction to counting problems also allows comparing the computational complexity of each level of the attacker hierarchy. Considering reachability objectives alone, we see that our approach, compared to classical reachability objectives, distinguishes between the different levels of the attacker hierarchy. This distinction is also present for *robust reachability* [GFB21] objectives, but these objectives currently do not allow to easily represent a leakage objective.

Table 15.1 shows an overview of the mentioned counting problems together with their complexity. We also differentiate between *stochastic* problems (e.g MЯ-SSAT, SSAT, and DSSAT) and *counting* problems (e.g MAX#SAT and DQMAX#SAT) and show the *shape* of the prefixes of the considered problems for easier comparison.

We can see that defining security as quantitative (or robust) reachability seems to fit our attacker model, given that this criterion for security allows to differentiate between classes

	Non-adaptive	Adaptive	Multiple Adaptive
Classic reachability	$\exists^*\phi$	$\exists^*\phi$	$\exists^*\phi$
Boolean problems	NP (SAT)	NP (SAT)	NP (SAT)
Robust reachability	$\exists^*\forall^*\phi$	$(\forall + \exists)^*\phi$	$\forall^*(\exists^H)^*\phi$
Boolean problems	Σ_2	PH (QBF)	NExpTime (DQBF)
	$\mathsf{M}^*\mathsf{Я}^*\phi$	$(\mathsf{Я} + \mathsf{M})^*\phi$	$\mathsf{Я}^*(\mathsf{M}^H)^*\phi$
Stochastic problems	$\mathsf{NP}^{\#\cdot\mathsf{P}}$ (MЯ-SSAT)	CH (SSAT)	NExpTime (DSSAT)
Quantitative reachability	$\mathsf{M}^*\mathsf{Я}^*\exists^*\phi$	-	$\mathsf{Я}^*\exists^*(\mathsf{M}^H)^*\phi$
Counting problems	$\mathsf{NP}^{\#\cdot\mathsf{NP}}$ (MAX#SAT)	-	NExpTime (DQMAX#SAT)

Table 15.1: Overview of the problems encountered

of attackers in terms of computational complexity. On the other hand, classic reachability cannot distinguish between the different levels of the attacker hierarchy.

Following this method, and motivated by security evaluation in practice, we studied each counting problem independently. This leads to new resolution algorithms in the case of two of them: Max#SAT and DQMax#SAT.

For Max#SAT and non-adaptive attacks, a new algorithm is proposed. This algorithm takes a model counting oracle (either exact or approximate) as input, and synthesizes the optimal answer for the attacker. Due to the prevalence of approximate model counters [CMV13; MA20], we showed that our algorithm offers good guarantees in the approximate setting. This allows to have resolution in practice, as demonstrated by our experimental evaluation.

For DQMax#SAT and multiple adaptive attacks, we introduced this new problem as well as several resolution methods. We also provided proofs of its computational complexity by reduction to existing well-known problems. Among our resolution methods we implemented one called *incremental* and experimented it against a set of benchmarks extracted from security examples.

Finally for F-SSAT and adaptive attacks, our study of the existing solvers showed that exact solving does not seem to be practical at the time of writing.

15.2 Future works and perspectives

We finally present the current areas of work around the contributions presented in this thesis, as well as possible extensions.

15.2.1 Integration in program analysis tools

A main area of work is integrating the framework that we developed into an existing program analysis tool. For that regard, we have done early work using the `binsec` [Dav+16] tool, with some success. A difficulty is that program analysis tools operate on bit-vectors together with arrays to represent the memory. This means that the normalisation process described in Chapter 5 needs to also account for arrays. In order to have a complete integration it would thus be needed to exclude the presence of arrays in the program, which can be done through techniques like read-over-write elimination and ackermanisation. Also, as mentionned in Chapter 4, SIP allows to represent multiple program paths in one sub-program, and perform security evaluation on that sub-program. This concretely amounts to *path-merging* within the symbolic execution tool, which we plan to work on.

Furthermore, `binsec` does not yet have primitives for input/output handling, therefore limiting the range of the attacker hierarchy to only non-adaptive attacks. We aim to extend the tool to include input/output primitives, as well as channel-based communication (Chapter 11), thus allowing to use the complete range of the attacker hierarchy.

15.2.2 Improving Max#SAT resolution

From an algorithmic point of view, the method presented in Chapter 7 could be extended in several directions.

First, we exploited some classes of symmetries when solving Max#SAT (Section 7.4). This could be improved by detecting new kinds of symmetries [Dev+16], or exploiting them further using techniques such as symmetry propagation [Met+18].

As discussed in Section 7.3, our relaxation algorithm (Algorithm 2) uses a *linear sweep* over the literals composing a witness. Instead of returning one possible minimal relaxation, `MergeXPlain` [SJS15] returns multiple ones, which may be helpful in our case by allowing the creation of multiple blocking clauses.

As expected, in some instances, our algorithm may degenerate into exhaustive search. While we do not know yet any characterization of all such instances, we believe that preprocessing and in-processing [JHB12] techniques such as `unhiding` [HJB11] should improve performance and limit the set of inefficient instances.

Finally, Algorithm 1 may be parallelized by correctly scheduling search spaces among threads, possibly using the leads described in Section 7.5.2. If we enforce the fact that all leads currently present in the lead list are disjoint, that is the $[\tilde{x}]_E$ are pairwise disjoint (hence splitting the search space into parts), we could expect a favorable parallelization setting.

15.2.3 Improving DQMax#SAT resolution

The resolution method presented in Chapter 13 can be expanded in several directions.

First, we would like to enhance our prototype with strategies for dependency expansion in the incremental algorithm. Experiments showed that our method's performances depend heavily on the number of dependencies in each dependency set. Techniques from DSSAT [LCJ23; LJ21] could lower the size of the dependency sets and thus mitigate the performance issues.

Second, we plan to integrate the local resolution method in our prototype. The local method exploits cases where DQMax#SAT resolution for a given dependency boils down to SSAT resolution. Using this fact, we could enhance our algorithm to use the techniques mentioned in the previous section to allow for approximate resolution of DQMax#SAT.

15.2.4 Approximate resolution for SSAT

At the time of writing, there are no approximate resolution method for SSAT. As with other counting problems, approximate resolution allows having a trade-off between resolution time and correctness of the returned answer.

While some solvers return bounds on the optimal answer found thus far [CHJ21], having *probably approximately correct* (PAC) bound, as well as some guarantees about the runtime of the algorithm based on the approximation parameters would be an important improvement.

An issue with SSAT is quantifier alternation, resulting in two kinds of sub-problems to be solved: optimisation problems (M levels) and *discrete integration* problems (R levels). We believe that combining techniques used for Max#SAT for optimisation levels with approximation schemes for discrete integration [Erm+13] can give a reasonable approximation scheme for SSAT with PAC guarantees. Implementing this algorithm, as well as comparing it with state of the art solvers would push the synthesis of adaptive attacks forward.

15.2.5 Counting problems with projections

Throughout this thesis, as presented in Table 15.1, we presented multiple counting problems, namely #SAT, #∃SAT, SSAT, DSSAT and DQMax#SAT. Of those, two pairs of problems stand out: $\mathsf{M}\text{Я}$-SSAT and MAX#SAT, and DSSAT and DQMAX#SAT. In both cases the second appears to be strictly more general that the first.

The last two rows of Table 15.1 indicate perspectives of interest for potential future work. When presenting optimal attack synthesis, we showed that counting problems allow succinct encoding of the attack synthesis: (i) MAX#SAT for non-adaptive attackers, (ii) DQ-MAX#SAT for multiple adaptive attackers. For adaptive attackers, we presented an encoding using SSAT. This encoding is still succinct, but:

1. following the semantics of SSAT, it implies computing the probability of the sub-formulas for the entire prefix, including the last maximizing-preffixed set of variable

2. based on the prefix construction, it does not allow an easy differentiation between the *roles* of each variables within the program (e.g. outputs, inputs, and random variables)

We know that optimal attack synthesis for adaptive attackers also fits nicely in DQ-MAX#SAT, where the dependency sets will admit an order in the sense of set inclusion. This strict subset of DQMAX#SAT problems is obviously a generalization of SSAT, allowing to use *projections* much like MAX#SAT and DQMAX#SAT.

A first approximation of this problem can be problems where prefixes are of the form: $(\text{Я} + \mathsf{M})^*\exists^*$. In this case, we can slightly modify the definition of SSAT to allow for projection *at the innermost levels of quantifiers*.

However, at the time of writing, we are still investigating formal definitions of the more general problem and its semantics. Note that any progress in formalising (and thus solving) this new problem will eventually allow for better optimal strategy synthesis for adaptive attackers.

Furthermore, the difference in computational complexity between the last two lines of Table 15.1 seem to indicate that a gap exists (see Remark 3.2.1) between counting problems (with projection) and stochastic problems (without projection).

Appendix A

Éléments de traduction en français

Cette annexe contient une traduction du manuscrit en français. La Section A.1 contient un résumé global de la thèse, avec son contexte et ses contributions. La Section A.2 donne un résumé de chaque chapitre. Enfin la Section A.3 propose une conclusion pour la thèse.

A.1 Résumé global

A.1.1 Introduction

Évaluer la sécurité d'un programme est une tâche particulièrement difficile, mais d'une importance capitale étant donné l'importance des systèmes informations dans le monde d'aujourd'hui.

Une manière possible pour l'évaluation de la sécurité se fait par étapes : d'abords chercher des vulnérabilités et des bugs dans le programme, et ensuite évaluer comment transformer ces bugs en profits, que ce soit des informations ou de nouvelles capacités par rapport au programme. Pour évaluer la sécurité d'un programme, il est donc nécessaire de considérer un modèle d'attaquant. Nous proposons dans cette thèse une hiérarchie d'attaquants, avec des capacités de raisonnement grandissantes à mesure que l'on croît dans la hiérarchie : non-adaptatif, adaptatif, et plusieurs attaquants adaptatifs. Nous étudions les niveaux de la hiérarchie en variant l'objectif de l'attaquant, que ce soit gagner des informations ou déclencher des bugs.

Alors que des critères qualitatifs de sécurité des programmes existent, leurs pendants quantitatifs sont relativement peu étudiés. L'intérêt des critères quantitatifs est double : il est possible de comparer des programmes de manière à, par exemple, vérifier si une vulnérabilité a bien été corrigée, ou ordonner les bugs à corriger en priorité ; mais il est aussi possible de fixer un objectif quantitatif que le programme doit atteindre, et qui dépend du domaine d'application, et appliquer une politique de sécurité de cette manière.

L'objectif de cette thèse est donc de développer des nouveaux objectifs quantitatifs pour la sécurité des programmes, basés sur l'objectif de l'attaquant, ce qui donne des problèmes de comptage. Les problèmes de comptage sont, comme leur nom l'indique, des problèmes liés au calcul de la taille d'un ensemble. Nous concentrons notre intérêt sur un problème de comptage par niveau de la hiérarchie : MAX#SAT pour les non-adaptatifs, SSAT pour les adaptatifs, et DQMAX#SAT pour les coalitions. Dans cette thèse, nous définissions formellement deux objectifs pour les attaquants, et développons des encodages desdits objectifs comme des problèmes de comptage. Nous résolvons ensuite ces problèmes d'optimisation combinatoire, qui correspondent à répondre à la question : ”à quel point est-il facile pour l’attaquant de satisfaire son objectif ?”.

A.1.2 Problèmes

Nous pouvons donc distinguer les problèmes suivants, d’intérêt dans le cadre de cette thèse :

Les attaquants avancés Pour définir une notion de *sécurité* pour les programmes, il est nécessaire de prendre en considération un *modèle d’attaquant.* Les modèles d’attaquants sont des objets formels qui représentent les capacités de l’attaquant. Une méthode d’analyse de programmes est évidement liée à un modèle d’attaquant donné. Plusieurs capacités orthogonales peuvent être associées pour construire un modèle d’attaquant :

1. quelle est sa condition de victoire ? Doit-il atteindre un point donné du programme ? Découvrir la valeur d’un secret ?

2. À quel point l’attaquant peut-il influencer l’état du programme ? Peut-il injecter des fautes ? Peut-il fournir des entrées au programme ?

3. Peut-il apprendre via des interactions avec le programme ?

Plusieurs exemples de modèles d’attaquants existent dans la littérature. Ducousso, Bardin et Potet [DBP23] présentent une nouvelle méthode d’exécution symbolique basée sur une modèle d’attaquant qui peut injecter des fautes dans le matériel durant l’exécution. Ils gardent cependant un objectif d’atteignabilité simple. Girol, Farinier et Bardin [GFB21] définissent une autre méthode d’exécution symbolique qui considère un attaquant construisant des valeurs d’entrée pour le programme et dont l’objectif est d’atteindre un point du programme de manière *robuste*, c’est-à-dire indépendamment des valeurs des secrets du programme. Enfin, Smith [Smi09] considère un attaquant dont l’objectif est de découvrir les secrets du programme.

La vérification de sécurité Évaluer la sécurité d’un programme revient à trouver un bogue dans celui-ci, puis à évaluer son *exploitabilité.* Malheureusement, l’exploitabilité est une définition floue dans la littérature [Dul20 ; Gir22 ; Smi09]. Afin de définir ce qu’est un programme sécurisé, il est nécessaire de d’abords définir ce que signifie l’exploitabilité. Implicitement, ceci implique des hypothèses sur le modèle d’attaquant considéré.

Une bonne façon d’évaluer la sécurité d’un programme est donc de définir un modèle d’attaquant d’abords, et ensuite de déterminer une stratégie gagnante pour ce modèle d’attaquant, face au programme. C’est ainsi une instance du problème dit de *synthèse.*

A.1.3 Propositions

Dans cette thèse, nous proposons un critère de sécurité basé sur les modèles d’attaquants, couplé avec des objectifs *quantitatifs.* Notre objectif est de montrer que la synthèse de la stratégie d’attaque optimale est liée à des problèmes connus, permettant la synthèse desdites stratégies en pratique. Nous formulons trois contributions principales afin de conduire l’évaluation de sécurité d’un programme.

Les *programmes interactifs simples* Nous définissons un langage impératif simple, conçu pour être une forme intermédiaire formelle pour les programmes, et sur laquelle il est possible de définir correctement et de manière concise la sécurité. Ce langage formel contient des constructions explicites pour la gestion d'interactions entre de programme et l'attaquant, permettant de suivre formellement ces interactions lors de l'exécution du programme. De plus, nous définissons la sémantique de ce langage de programmation en nous concentrant sur l'évaluation quantitative de la sécurité par rapport à l'attaquant.

Objectifs d'attaquants quantitatifs Nous proposons une formulation des objectifs d'attaquants basée sur les probabilités, qui unifie les notions d'atteignabilité quantifiée [Gir22] et de fuite d'information [Smi09]. Unifier ces définitions permets leur traitement commun au sein d'un même cadre formel. De plus, nous montrons que ces propriétés de sécurités quantitatives se réduisent à des problèmes dits *de comptage* [Tod91 ; Tor91], une classe de problèmes en lien avec le décompte du nombre d'éléments dans un ensemble et leur généralisation à des problèmes d'optimisation combinatoire.

La hiérarchie d'attaquants De manière orthogonale, nous définissons une hiérarchie de modèles d'attaquant variant en *capacité de raisonnement*. Nous considérons trois niveaux dans cette hiérarchie :

1. **non-adaptatif** : l'attaquant n'a pas le droit d'exploiter les sorties de programmes pour adapter sa stratégie. Il est uniquement autorisé à fournir ses entrées *a priori* de l'exécution.
2. **adaptatif** : l'attaquant est autorisé à interagir avec le programme et apprendre de ses interactions. Les entrées que l'attaquant fournit peuvent donc dépendre des sorties précédentes du programme et deux exécutions différentes peuvent mener à deux stratégies différentes.
3. **coalition adaptative** : ce niveau représente plusieurs attaquants qui partagent un objectif commun, mais qui ne partagent pas forcément d'information durant l'exécution du programme. Dans ce cas, la stratégie optimale de la coalition n'est pas nécessairement la combinaison des stratégies optimales individuelles.

A.1.4 Contributions

Nous ordonnons les contributions de cette thèse en suivant l'organisation des différents chapitres :

- la Partie II introduit notre formalisme pour les programmes, ainsi que notre nouveau cadre formel pour unifier les objectifs basés sur la fuite d'information et sur l'atteignabilité quantifiée. Le Chapitre 5 présente des techniques de normalisation des programmes pour le langage introduit pour éviter les problèmes d'explosion de chemins. Le Chapitre 11 propose une extension de ce langage pour les coalitions.
- les Parties III à V étudient chacune un niveau de la hiérarchie d'attaquants (respectivement non-adaptatif, adaptatif, et les coalitions). Chaque partie suit la même organisation :

1. Nous introduisons d'abords les particularités de la synthèse d'attaque optimale à ce niveau, ainsi que le problème de comptage correspondant. Nous prouvons ensuite l'équivalence entre la synthèse d'une attaque optimale et le problème de comptage.
2. Ensuite, nous proposons une méthode de résolution pour le problème de comptage. Ces méthodes de résolutions sont conçues pour la résolution en pratique, et se basent sur des schémas d'approximation pour atteindre cet objectif.
3. Enfin, nous évaluons expérimentalement la méthode de résolution proposée, sur des exemples extraits de problèmes de sécurité.

A.2 Résumé des chapitres

A.2.1 Partie II : programmes et attaques

Cette partie introduit le langage impératif simple (SIP) qui sera utilisé au long de la thèse afin de représenter les programmes et évaluer la sécurité. Nous introduisons aussi une définition pour les différents objectifs d'attaques, ainsi qu'une manière de les évaluer.

Chapitre 4 : les programmes interactifs simples

Ce chapitre donne la définition formelle du langage formel. Celui-ci sert comme une *étape intermédiaire* dans l'analyse des programmes. Il peut être vu comme un cadre formel entre l'évaluation de la sécurité et les problèmes de comptage. Son but est d'être assez expressif pour représenter le comportement du *programme source* de manière concise, mais aussi assez simple pour permettre des définitions simples des problèmes de sécurité.

Nous choisissons un langage simple qui gère les entrées/sorties de manière explicite, ainsi qu'un modèle d'exécution strict., Ceci permet à notre langage d'être une représentation concise pour les exécutions du programme source atteignant un point donné du programme. Cela signifie qu'il est nécessaire d'analyser ledit programme pour chercher les chemins atteignant le point ciblé, et ensuite les représenter sur la forme d'un programme SIP, qui permettra de répondre à la question de la facilité éventuelle qu'un attaquant exploite l'ensemble de chemins sélectionné.

Chapitre 5 : normaliser des programmes SIP

Dans ce chapitre, notre objectif est de transformer des programmes SIP vers une forme normale qui préserver certaines propriétés du programme original, tout en simplifiant leur étude. Les différentes transformations impliquées sont discutées une par une, et cela permet de ne considérer que des programmes en forme normale dans le reste de la thèse.

Nous commençons par une notion d'équivalence de programmes qui est d'importance dans ce chapitre. Notre objectif est de prouver qu'un programme est équivalent à sa transformation pour toutes les transformations mentionnées. Nous voulons également montrer que cette relation implique la préservation à la fois de l'atteignabilité quantifiée, mais aussi de la fuite d'information du programme.

A.2.2 Partie III : les attaquants non-adaptatifs

Dans cette partie, nous étudions le premier cas de la hiérarchie d'attaquants : les attaquants non-adaptatifs. Cette partie suit le plan exposé précédemment : réduction du problème de synthèse d'une attaque optimale vers un problème de comptage (Max#SAT) ; proposition d'une méthode de résolution du problème ; et évaluation expérimentale de la solution proposée.

Chapitre 6 : réduction à Max#SAT

Ce chapitre présente le problème de compte Max#SAT et la réduction de la synthèse d'une attaque optimale vers celui-ci. Les attaques non-adaptative sont, informellement, celles qui n'utilisent pas les informations reçues depuis le programme cible pour adapter leurs stratégies.

Chapitre 7 : résolution de Max#SAT

Nous proposons dans ce chapitre in algorithme pour la résolution de Max#SAT, motivé par la réduction présentée préalablement. Ce problème demande, étant donné une formule booléenne, d'optimiser, par rapport à un certain sous ensembles de variables (les variables *maximisantes*) de cette formule, le nombre de modèles du co-produit de la formule sur un autre ensemble de variables, les variables *de comptage*. Ce problème est d'intérêt dans notre cas étant donné les réductions mentionnées plus tôt.

Malheureusement, Max#SAT est d'une grande complexité calculatoire [Tor91], et les méthodes de résolution restent coûteuses. Ceci motive l'utilisation de schémas d'approximation de façon à résoudre le problème en temps et en espace raisonnable, tout en garantissant une certaine correction du résultat.

Nous proposons donc un algorithme pour résoudre ce problème, basé sur des techniques dites de *raffinement d'abstraction guidé par les contre-exemples* (CEGAR). Cet algorithme admet deux modes de fonctionnements en fonctions des oracles utilisés : un mode exact, donnant la réponse optimale à coup sûr ; et un mode approché, donnant une réponse arbitrairement proche de la réponse optimale.

Chapitre 8 : évaluation expérimentale

Nous évaluons, dans ce chapitre, l'efficacité de l'algorithme présenté au chapitre précédent.

Pour cela, nous utilisons des cas d'études de l'état de l'art sur le sujet, afin de nous comparer avec d'autres algorithmes existants. Nous montrons que notre algorithme permet une résolution plus rapide des cas d'exemple par rapport à l'état de l'art.

A.2.3 Partie IV : les attaquants adaptatifs

Dans cette partie, nous étudions les attaquants adaptatifs. Nous proposons, dans ce cas, uniquement une réduction vers un problème de comptage connu (SSAT), ainsi qu'une évaluation expérimentale de cette réduction.

Chapitre 9 : réduction à SSAT

Nous réduisons dans ce chapitre le problème de la synthèse d'attaque adaptative au problème de comptage connu SSAT.

Les adaptatives sont évidemment des généralisations strictes des attaques non-adaptatives, et ainsi, le problème correspondant est, lui aussi, une généralisation stricte. Nous prouvons donc des résultats similaires aux réductions précédentes, en suivant le schéma de la Partie III.

Chapitre 10 : évaluation expérimentale

Nous évaluons expérimentalement notre approche dans ce chapitre, en utilisant des solveurs SSAT de l'état de l'art. Nous réduisons les exemples mentionnés au long de la thèse afin de les soumettre aux solveurs, ce qui nous permet de démontrer la faisabilité de notre approche.

A.2.4 Partie V : coalitions d'attaquants

Dans cette partie, nous étudions le dernier échelon de notre hiérarchie d'attaquants : les coalitions. Ces coalitions représentent des attaquants avec un objectif commun, mais ne partageant pas d'information au cours de l'attaque.

Chapitre 11 : SIP multi-canal

Nous introduisons dans ce chapitre une variante plus générale du langage SIP, permettant la communication à travers plusieurs canaux. Ainsi, les programmes m-SIP sont strictement plus expressifs que les programmes SIP à travers l'usage de ces canaux (SIP n'est utilisé qu'un). Cependant, les capacités individuelles des attaquants au sein de la coalition sont diminuées : ceux-ci ne peuvent voir que les informations apparaissant sur leur canal. Malheureusement, la complexité de la synthèse d'une attaque optimale est plus grande, car il est nécessaire de considérer toutes les combinaisons possibles des stratégies des acteurs de la coalition.

Intuitivement, cette généralisation revient à ajouter un paramètre d'*identifiant de canal* pour les primitives de communication. Nous voyons dans ce chapitre que le problème de comptage correspondant à cette catégorie d'attaque est différent et plus complexe calculatoirement.

Nous proposons, comme pour la Partie II, normalisation de ces programmes afin de faciliter leur étude. Cette normalisation permet ensuite la réduction vers le problème de comptage d'intérêt dans cette partie.

Chapitre 12 : DQMax#SAT

Dans ce chapitre, nous introduisons le problème de comptage correspondant à la synthèse de stratégies d'attaque optimale pour les coalitions.

Nous utilisons pour cela des *quantificateurs de Henkin* [HK65]. Informellement, ces quantificateurs permettent de spécifier explicitement les dépendances entre les variables dans le préfixe de la formule. À l'opposé des *quantificateurs classiques* (c'est-à-dire $\forall$ et $\exists$ en logique du premier ordre), les quantificateurs de Henkin permettent de spécifier des ordres partiels (non-linéaires) entre les dépendances de variables. Ceci est d'intérêt étant donné la différence entre SIP et m-SIP, le premier nécessitant un ordre linéaire dans les dépendances entre les événements, et l'autre permettant des ordres partiels en général.

Chapitre 13 : résoudre DQMax#SAT

Nous présentons dans ce chapitre plusieurs méthodes de résolution pour le problème DQMax#SAT. Nous étudions également la complexité computationelle du problème, en le situant par rapport à d'autres problèmes existants.

La première méthode de résolution est une réduction globale vers Max#SAT. Celle-ci souffre cependant d'une explosion exponentielle dans la taille de la formule, en lien avec la taille des ensembles de dépendances.

La deuxième méthode de résolution est un raffinement de la première et dite *incrémentale*. Celle-ci se base sur une construction progressive de la formule Max#SAT, en considérant des ensembles de dépendances de plus en plus grands. Cette méthode permet la synthèse d'une *solution préliminaire*, qui permet une approximation de la stratégie optimale.

La dernière méthode est une méthode de simplification du problème, s'appliquant uniquement dans certains cas. Cette méthode exploite des schémas similaires à ceux correspondants aux attaques adaptatives, et permet la réduction de la résolution d'un problème à un ensemble de sous-problèmes plus simples. Cela permet de limiter l'explosion de la taille de la formule dans les cas d'application.

Chapitre 14 : évaluation expérimentale

Nous évaluons expérimentalement l'algorithme incrémental de résolution de DQMax#SAT dans ce chapitre.

Cette évaluation est faite en considérant deux solveurs possibles pour les requêtes Max#SAT : l'algorithme présenté dans cette thèse et un algorithme exact de l'état de l'art.

Nous soumettons à un outil prototype des exemples basés sur les exemples de sécurité de cette thèse, et nous montrons que la complexité de ce problème est une limite majeure pour la résolution.

A.3 Conclusion

Durant cette thèse, nous avons développé une méthode pour évaluer quantitativement la sécurité des programmes. Ce nouveau cadre formel requiert la synthèse d'une stratégie d'attaque optimale par rapport à un objectif, unifiant et étendant à la fois l'atteignabilité quantifiée [BG22] et la fuite d'information [Smi09].

Ce cadre unifié permet aussi différents *types* d'attaquants, organisé dans une hiérarchie : non-adaptatifs, adaptatifs, et coalitions. Nous avons montré que la synthèse d'une attaque optimale à chaque niveau de la hiérarchie correspond à un *problème de comptage* particulier, respectivement : Max#SAT, f-SSAT et DQMax#SAT. La réduction vers des problèmes de comptage permets aussi de comparer la complexité computationelle à chaque niveau de la hiérarchie d'attaquants. En considérant uniquement l'atteignabilité quantifiée, remarquons que notre approche, comparée à des objectifs d'atteignabilité classique, distingue les différents niveaux de la hiérarchie. Cette distinction est aussi présente pour l'atteignabilité robuste [GFB21], mais ce formalisme ne permet pas de prendre en compte la fuite d'information.

Suivant cette méthode, et motivé par la résolution en pratique, nous avons étudié chacun des problèmes de comptage indépendamment. Cela à mené à de nouvelles méthodes de résolution dans le cas de deux d'entre eux : Max#SAT et DQMax#SAT.

Nomenclature

Boolean logic notations

$\mathbb{B}$	The set of boolean values
$\perp$	Boolean value *false*
$\mathcal{F}\langle E\rangle$	The set of boolean formulas with variables in E
$\mathcal{M}\langle E\rangle$	The set of complete monomials over E
$\top$	Boolean value *true*

Probability theory notations

$\mathbb{D}(E)$	The set of distributions over E
$\mathbb{I}[H, L]$	The mutual information between distribution H and L
$\mathbb{P}[e]$	The probability that an event e occurs
$\mathbb{H}_\alpha[D]$	The Rényi entropy of order α of D
$\mathbb{H}_S[D]$	The Shannon entropy of a distribution D
$\mathbb{V}[D]$	The vulnerability of a distribution D

Mathematical notations

$\lvert\cdot\rvert$	Cardinal of a set

Bibliography

[ABB15] Abdulbaki Aydin, Lucas Bang and Tevfik Bultan. “Automata-Based Model Counting for String Constraints”. In: *CAV (1)*. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 255–272.

[ALM22] Gilles Audemard, Jean-Marie Lagniez and Marie Miceli. “A New Exact Solver for (Weighted) Max#SAT”. In: *SAT*. Vol. 236. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 28:1–28:20.

[ASU86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[Aud+22] Gilles Audemard, Jean-Marie Lagniez, Marie Miceli and Olivier Roussel. “Identifying Soft Cores in Propositional Formulæ”. In: *ICAART (2)*. SCITEPRESS, 2022, pp. 486–495.

[Azi+15] Rehan Abdul Aziz, Geoffrey Chu, Christian J. Muise and Peter J. Stuckey. “Projected Model Counting”. In: *CoRR* abs/1507.07648 (2015).

[BG22] Sébastien Bardin and Guillaume Girol. “A Quantitative Flavour of Robust Reachability”. In: *CoRR* abs/2212.05244 (2022).

[CC77] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *POPL*. ACM, 1977, pp. 238–252.

[CDE08] Cristian Cadar, Daniel Dunbar and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *OSDI*. USENIX Association, 2008, pp. 209–224.

[Cha+15] Supratik Chakraborty, Dror Fried, Kuldeep S. Meel and Moshe Y. Vardi. “From Weighted to Unweighted Model Counting”. In: *IJCAI*. AAAI Press, 2015, pp. 689–695.

[CHJ21] Pei-Wei Chen, Yu-Ching Huang and Jie-Hong R. Jiang. “A Sharp Leap from Quantified Boolean Formula to Stochastic Boolean Satisfiability Solving”. In: *AAAI*. AAAI Press, 2021, pp. 3697–3706.

[CMV13] Supratik Chakraborty, Kuldeep S. Meel and Moshe Y. Vardi. “A Scalable Approximate Model Counter”. In: *CoRR* abs/1306.5726 (2013).

[Dav+16] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet and Jean-Yves Marion. “BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis”. In: *SANER*. IEEE Computer Society, 2016, pp. 653–656.

[DBP23] Soline Ducousso, Sébastien Bardin and Marie-Laure Potet. “Adversarial Reachability for Program-level Security Analysis”. In: *ESOP*. Vol. 13990. Lecture Notes in Computer Science. Springer, 2023, pp. 59–89.

[Dev+16] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe and Marc Denecker. “Improved Static Symmetry Breaking for SAT”. In: *SAT*. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 104–122.

[DHK00] Arnaud Durand, Miki Hermann and Phokion G. Kolaitis. “Subtractive Reductions and Complete Problems for Counting Complexity Classes”. In: *MFCS*. Vol. 1893. Lecture Notes in Computer Science. Springer, 2000, pp. 323–332.

[DM11] Adnan Darwiche and Pierre Marquis. “A Knowledge Compilation Map”. In: *CoRR* abs/1106.1819 (2011). URL: `https://arxiv.org/pdf/1106.1819.pdf`.

[DP60] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (1960), pp. 201–215.

[Dul20] Thomas Dullien. “Weird Machines, Exploitability, and Provable Unexploitability”. In: *IEEE Trans. Emerg. Top. Comput.* 8.2 (2020), pp. 391–403. URL: `https://ieeexplore.ieee.org/ielx7/6245516/9109372/08226852.pdf`.

[Erm+13] Stefano Ermon, Carla P. Gomes, Ashish Sabharwal and Bart Selman. “Taming the Curse of Dimensionality: Discrete Integration by Hashing and Optimization”. In: *ICML (2)*. Vol. 28. JMLR Workshop and Conference Proceedings. JMLR.org, 2013, pp. 334–342.

[FJ23] Yu-Wei Fan and Jie-Hong R. Jiang. “SharpSSAT: A Witness-Generating Stochastic Boolean Satisfiability Solver”. In: *AAAI*. AAAI Press, 2023, pp. 3949–3958.

[FRS17] Daniel J. Fremont, Markus N. Rabe and Sanjit A. Seshia. “Maximum Model Counting”. In: *AAAI*. AAAI Press, 2017, pp. 3885–3892.

[GFB21] Guillaume Girol, Benjamin Farinier and Sébastien Bardin. “Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference”. In: *CAV (1)*. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 669–693.

[Gir22] Guillaume Girol. “Robust reachability and model counting for software security”. 2022UPASG071. PhD thesis. 2022. URL: `http://www.theses.fr/2022UPASG071/document`.

[GMK15] M. Gario, A. Micheli and Bruno Kessler. “PySMT : a Solver-Agnostic Library for Fast Prototyping of SMT-Based Algorithms”. In: 2015.

[Gol+21] Priyanka Golia, Mate Soos, Sourav Chakraborty and Kuldeep S. Meel. “Designing Samplers is Easy: The Boon of Testers”. In: *FMCAD*. IEEE, 2021, pp. 222–230.

[HJB11] Marijn Heule, Matti Järvisalo and Armin Biere. “Efficient CNF Simplification Based on Binary Implication Graphs”. In: *SAT*. Vol. 6695. Lecture Notes in Computer Science. Springer, 2011, pp. 201–215.

[HK65] Leon Henkin and Carol R. Karp. "Some Remarks on Infinitely Long Formulas". In: *Journal of Symbolic Logic* 30.1 (1965), pp. 96–97. DOI: 10.2307/2270594.

[HV95] Lane A. Hemaspaandra and Heribert Vollmer. "The satanic notations: counting classes beyond #P and other definitional adventures". In: *SIGACT News* 26.1 (1995), pp. 2–13. URL: https://dl.acm.org/doi/pdf/10.1145/203610.203611.

[JHB12] Matti Järvisalo, Marijn Heule and Armin Biere. "Inprocessing Rules". In: *IJCAR*. Vol. 7364. Lecture Notes in Computer Science. Springer, 2012, pp. 355–370.

[JK07] Tommi A. Junttila and Petteri Kaski. "Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs". In: *ALENEX*. SIAM, 2007.

[Jun04] Ulrich Junker. "Preference-based Problem Solving for Constraint Programming". In: *Preferences*. Vol. 04271. Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany, 2004.

[Kul21] Oliver Kullmann. "Fundaments of Branching Heuristics". In: *Handbook of Satisfiability*. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 351–390.

[LCJ23] Yun-Rong Luo, Che Cheng and Jie-Hong R. Jiang. "A Resolution Proof System for Dependency Stochastic Boolean Satisfiability". In: *J. Autom. Reason.* 67.3 (2023), p. 26.

[LGM98] Michael L. Littman, Judy Goldsmith and Martin Mundhenk. "The Computational Complexity of Probabilistic Planning". In: *CoRR* cs.AI/9808101 (1998).

[LJ21] Nian-Ze Lee and Jie-Hong R. Jiang. "Dependency Stochastic Boolean Satisfiability: A Logical Formalism for NEXPTIME Decision Problems with Uncertainty". In: *AAAI*. AAAI Press, 2021, pp. 3877–3885.

[LM17] Jean-Marie Lagniez and Pierre Marquis. "An Improved Decision-DNNF Compiler". In: *IJCAI*. ijcai.org, 2017, pp. 667–673.

[LM19] Jean-Marie Lagniez and Pierre Marquis. "A Recursive Algorithm for Projected Model Counting". In: *AAAI*. AAAI Press, 2019, pp. 1536–1543.

[LMY21] Yong Lai, Kuldeep S. Meel and Roland H. C. Yap. "The Power of Literal Equivalence in Model Counting". In: *AAAI*. AAAI Press, 2021, pp. 3851–3859.

[LWJ18] Nian-Ze Lee, Yen-Shi Wang and Jie-Hong R. Jiang. "Solving Exist-Random Quantified Stochastic Boolean Satisfiability via Clause Selection". In: *IJCAI*. ijcai.org, 2018, pp. 1339–1345.

[MA20] Kuldeep S. Meel and S. Akshay. "Sparse Hashing for Scalable Approximate Model Counting: Theory and Practice". In: *CoRR* abs/2004.14692 (2020).

[MB05] Stephen M. Majercik and Byron Boots. "DC-SSAT: A Divide-and-Conquer Approach to Solving Stochastic Satisfiability Problems Efficiently". In: *AAAI*. AAAI Press / The MIT Press, 2005, pp. 416–422.

[Mes19] David Mestel. "Quantifying information flow in interactive systems". In: *CoRR* abs/1905.04332 (2019).

[Met+18] Hakan Metin, Souheib Baarir, Maximilien Colange and Fabrice Kordon. "CD-CLSym: Introducing Effective Symmetry Breaking in SAT Solving". In: *TACAS (1)*. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 99–114.

[MJB13] João Marques-Silva, Mikolás Janota and Anton Belov. "Minimal Sets over Monotone Predicates in Boolean Formulae". In: *CAV*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 592–607.

[Mon22] David Monniaux. "$NP^{\#P} = \exists PP$ and other remarks about maximized counting". In: *CoRR* abs/2202.11955 (2022).

[Mos+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang and Sharad Malik. "Chaff: Engineering an Efficient SAT Solver". In: *DAC*. ACM, 2001, pp. 530–535.

[MU21] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *CADE*. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 625–635.

[OHe20] Peter W. O'Hearn. "Incorrectness logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 10:1–10:32.

[Pap85] Christos H. Papadimitriou. "Games Against Nature". In: *J. Comput. Syst. Sci.* 31.2 (1985), pp. 288–301.

[PR79] Gary L. Peterson and John H. Reif. "Multiple-Person Alternation". In: *FOCS*. IEEE Computer Society, 1979, pp. 348–363.

[PRA01] Gary Peterson, John Reif and Salman Azhar. "Lower bounds for multiplayer noncooperative games of incomplete information". In: *Computers & Mathematics with Applications* 41.7-8 (2001), pp. 957–992.

[Ric53] H. Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74 (1953), pp. 358–366.

[Sah+21] Seemanta Saha, William Eiers, Ismet Burak Kadron, Lucas Bang and T. Bultan. "Incremental Attack Synthesis". In: *ACM SIGSOFT Software Engineering Notes* 44 (2021), pp. 16–16.

[SBK05] Tian Sang, Paul Beame and Henry A. Kautz. "Heuristics for Fast Exact Model Counting". In: *SAT*. Vol. 3569. Lecture Notes in Computer Science. Springer, 2005, pp. 226–240.

[SGM20] Mate Soos, Stephan Gocht and Kuldeep S. Meel. "Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling". In: *CAV (1)*. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 463–484.

[Sho+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel and Giovanni Vigna. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2016, pp. 138–157.

[SJS15] Kostyantyn M. Shchekotykhin, Dietmar Jannach and Thomas Schmitz. "MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis". In: *IJCAI*. AAAI Press, 2015, pp. 3221–3228.

[Smi09] Geoffrey Smith. “On the Foundations of Quantitative Information Flow”. In: *FoS-SaCS*. Vol. 5504. Lecture Notes in Computer Science. Springer, 2009, pp. 288–302.

[SNC09] Mate Soos, Karsten Nohl and Claude Castelluccia. “Extending SAT Solvers to Cryptographic Problems”. In: *SAT*. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 244–257.

[Sto83] Larry J. Stockmeyer. “The Complexity of Approximate Counting (Preliminary Version)”. In: *STOC*. ACM, 1983, pp. 118–126.

[TF10] Tino Teige and Martin Fränzle. “Resolution for Stochastic Boolean Satisfiability”. In: *LPAR (Yogyakarta)*. Vol. 6397. Lecture Notes in Computer Science. Springer, 2010, pp. 625–639.

[Tod91] Seinosuke Toda. “Computational complexity of counting complexity classes”. PhD thesis. 東京工業大学, 1991.

[Tor91] Jacobo Torán. “Complexity Classes Defined by Counting Quantifiers”. In: *J. ACM* 38.3 (1991), pp. 753–774. URL: `https://dl.acm.org/doi/pdf/10.1145/116825.116858`.

[Tra19] Gregory Travis. *How the Boeing 737 Max disaster looks to a Software Developer*. Accessed: 22/07/2024. 2019. URL: `https://spectrum.ieee.org/how-the-boeing-737-max-disaster-looks-to-a-software-developer`.

[Val79] Leslie G. Valiant. “The Complexity of Enumeration and Reliability Problems”. In: *SIAM J. Comput.* 8.3 (1979), pp. 410–421. URL: `https://www.math.cmu.edu/~af1p/Teaching/MCC17/Papers/enumerate.pdf`.

[Vig+22] Thomas Vigouroux, Cristian Ene, David Monniaux, Laurent Mounier and Marie-Laure Potet. “BaxMC: a CEGAR approach to Max#SAT”. In: *FMCAD*. IEEE, 2022, pp. 170–178.

[Vig+23] Thomas Vigouroux, Marius Bozga, Cristian Ene and Laurent Mounier. “Function synthesis for maximizing model counting”. In: *CoRR* abs/2305.10003 (2023).

[Vig+24] Thomas Vigouroux, Marius Bozga, Cristian Ene and Laurent Mounier. “Function Synthesis for Maximizing Model Counting”. In: *VMCAI (1)*. Vol. 14499. Lecture Notes in Computer Science. Springer, 2024, pp. 258–279.

[Zha21] Hantao Zhang. “Combinatorial Designs by SAT Solvers”. In: *Handbook of Satisfiability*. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 819–858.

[Zhu+22] Xiaogang Zhu, Sheng Wen, Seyit Camtepe and Yang Xiang. “Fuzzing: A Survey for Roadmap”. In: *ACM Comput. Surv.* 54.11s (2022), 230:1–230:36.