FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Trace Reasoning in Formal Verification

Guiding Vampire in Induction

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Logic and Computation

eingereicht von

Pamina Georgiou, BSc
Matrikelnummer 01125496

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ.Prof. Dr.techn Laura Kovács, MSc
Mitwirkung: DI Bernhard Gleiss, BSc

Wien, 4. November 2019

Pamina Georgiou	Laura Kovács

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Trace Reasoning in Formal Verification

Guiding Vampire in Induction

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Logic and Computation

by

Pamina Georgiou, BSc
Registration Number 01125496

to the Faculty of Informatics
at the TU Wien

Advisor: Univ.Prof. Dr.techn Laura Kovács, MSc
Assistance: DI Bernhard Gleiss, BSc

Vienna, 4th November, 2019

Pamina Georgiou	Laura Kovács

Erklärung zur Verfassung der Arbeit

Pamina Georgiou, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. November 2019

Pamina Georgiou

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben.

Zuallererst gebührt mein Dank Dr. Laura Kovács, die meine Masterarbeit betreut hat und mir während des Masterstudiums bereits erlaubt hat mich im wissenschaftlichen Arbeiten in ihrer Arbeitsgruppe zu erproben. Für die Chancen und Förderungen, als auch für die Geduld und Hilfe bei der Betreuung dieser Arbeit möchte ich mich herzlich bedanken. Ebenso bedanke ich mich bei Dipl.Ing. Bernhard Gleiss, ohne den diese Arbeit nicht hätte entstehen können, für die oft augenöffnende Zusammenarbeit und die viele Zeit für intuitive Erklärungen am Whiteboard im Rahmen dieser Arbeit.

Ebenfalls möchte ich mich bei meinen “Temple” Studienkollegen und -kolleginnen bedanken, die mir mit Interesse, Hilfsbereitschaft und zahlreichen interessanten Debatten zur Seite standen. Außerdem möchte ich besonders Anna Maria Nau für das Korrekturlesen meiner Masterarbeit und den emotionalen Beistand in ausgesprochen harten Phasen danken.

Abschließend möchte ich mich insbesondere bei meiner Mutter und meinen Großeltern bedanken, die mir mein Studium durch ihre Unterstützung ermöglicht und mich in all meinen Entscheidungen liebevoll begleitet haben.

Kurzfassung

Automatisierte Softwareverifikation gewinnt durch die fortschreitende Digitalisierung zunehmend an Bedeutung. Unter diesem Licht widmet sich diese Masterarbeit der automatisierten Softwareverifikation mit Hilfe von automatisierten Beweissystemen in Prädikatenlogik.

Insbesondere werden Programme mit Arrays und Schleifen hinsichtlich funktionaler Korrektheit für einfache, als auch relationale Spezifikationen untersucht. Vor allem werden relationale Eigenschaften typischerweise verwendet, um Sicherheits- und Datenschutzgarantien in Anwendungen der Systemsicherheit zu formulieren. Dies erfordert oftmals Quantorenalternierung.

Für die Verifizierung dieser Eigenschaften werden imperative Programme auf ein Gültigkeitsproblem in eine neue Logik, die sogenannte Trace Logic, reduziert. Durch Reasoningaktivitäten über natürliche und ganze Zahlen, erfordern automatisierte Beweise in dieser Logik auch induktive Schlüsse, die nur schwer automatisch vollzogen werden können. In diesem Sinne widmet sich diese Arbeit der Formulierung geeigneter Lemmata, genannt Trace Lemmas, die das automatische Beweissystem bei Schlüssen, die Induktion erfordern, leitet. Das Ziel der Arbeit ist es geeignete Sets an Trace Lemmas für verschiedene Eigenschaften zu definieren und zu formulieren, sodass automatisierte Beweise mit Hilfe des VAMPIRE-Beweissystems generiert werden können ohne von programmspezifischen Annotationen durch Experten abhängig zu sein.

Abstract

This work is motivated by automating reasoning for program analysis and verification in full first-order logic. We are interested in reachability and relational properties about functional correctness of programs with loops and arrays. Particularly, relational properties are typically used to formulate security and privacy guarantees in applications of system security, and often require reasoning about first-order formulas with quantifier-alternations.

In our approach, we reduce the verification task of reachability and relational properties of imperative programs to a validity problem into a new logic, called trace logic, that is an expressive instance of first-order logic. Properties in trace logic involve reasoning about natural numbers and integers, and thus impose the burden of automating inductive reasoning in the full first-order setting of theorem proving.

We address this challenge by automatically instantiating a set of so-called trace lemmas that guide first-order provers in inductive reasoning. The aim of this thesis is to identify a "reasonable" set of sound trace lemmas that allow superposition-based automated theorem provers to prove inductive (non-)relational properties, without relying on user-provided program annotations like program-specific loop invariants. To discharge verification conditions we rely on the first-order theorem prover VAMPIRE.

Contents

CHAPTER 1

Introduction

1.1 Introduction

Software is an increasingly critical component of almost all areas of life. Be that in the professional world or in speaking to one's smart assistant at home, people are more and more willing to share their sensitive data. This gives rise to attacks exploiting any kind of bugs to willfully retrieve private information of others. While testing is a necessary means to assuring the functionality of software, it is often not enough to show that certain specifications are fully met. Hence, they do not provide a proof that software is actually doing what it should, nor that it fulfills requirements wrt. security, availability etc. Thus, the task of (automated) software verification is an ongoing research effort in computer science.

The aim of this endeavour is to formally prove programs bug-free. It is obvious that in this day of age the task of formally verifying programs and proving their correctness is of higher and higher importance. Given the increased degree of digitalization and use of software for sensitive data, recent bugs like Spectre [KHF+19] and Meltdown [LSG+18] highlight the importance of formal methods. To this end, automation becomes more and more important as growing codebases cannot be scaled to be proven manually with potentially thousands of changes a day. Hence it is evident that new methods and tools are needed that increase the degree of automation in software verification tasks that can be integrated during the developement of critical code.

While practically powerful, current automated approaches to software verification come with a number of limitations. They (i) are restricted in the logical expressiveness of the software properties they handle, that is only (decidable) first-order fragments, by a combination of SMT-solving and model checking, (ii) produce false positives due to over-approximation used for example in abstract interpretation and static analysis, (iii) require expert knowledge by providing loop invariants/assertions to handle inductive reasoning, as is the case e.g. in Dafny [Lei10] or (iv) need user guidance in proving software correct by using interactive theorem provers and hence do not scale well.

While there exists some work on automating induction such as [Lei12], many approaches just focus on a small set of programs or a decidable fragment. We address (some of) these limitations

and propose reasoning in trace logic $\mathcal{L}$ to analyze and prove properties over (sets of) sets of program traces. We express program semantics in $\mathcal{L}$ and use superposition-based first-order theorem proving to fully automate reasoning in many-sorted full first-order logic with equality. For doing so, we automatically instantiate a set of so-called *trace lemmas* that guide first-order provers in inductive reasoning.

In this thesis, we will introduce identified trace lemmas for different sets of program properties and illustrate example by example on how trace lemmas are used for inductive reasoning tasks of relational and non-relational program properties with the help of the superposition based prover VAMPIRE. The work is based on the program semantics given by the RAPID-tool that can be found in [BEG+19] and will also be given as background information in Section 2.3.

Contributions The main contributions are summarized below:

1. *Chapter 3.* We identified trace lemmas for reachability properties of programs containing arrays and loops. Specifically we found a set of general inductive lemmas to automatically prove properties that handle inductive reasoning over loops without depending on program-specific loop invariants or user-defined annotations. We discharge these verification conditions with the superposition-based VAMPIRE theorem prover.

2. *Chapter 4.* We extended this approach to so-called hyperproperties, that is relational properties over multiple program traces of a program, and defined inductive trace lemmas over time point and trace reasoning for security properties such as noninterference and sensitivity.

3. The automatic instantiation of these lemmas is implemented in the software verification tool RAPID[1]. Together with its existing program semantics as defined in Chapter 2, RAPID generates verification conditions including instantiations of trace lemmas that can be automatically discharged with VAMPIRE.

1.2 Structure of the Thesis

The thesis will be structured as follows: After an overview of our method of using VAMPIRE for software verification with RAPID as well as an introduction of the syntax and semantics of our intermediate language and its encoding in first-order logic given in Chapter 2, we will dive into software verification and particularly the trace reasoning for handling inductive reasoning tasks involved in the process. Chapter 3 will focus on a standard non-relational setting, that is we will show how we prove program properties as sets of traces with the help of so-called trace lemmas in the superposition calculus. In Chapter 4, we will extend this approach to a more general setting, allowing us to handle multiple traces and prove relational properties, that is program properties as sets of sets of traces. As such we study particularly the trace reasoning that allows us to prove so-called hyperproperties like noninterference and sensitivity. We illustrate this approach on multiple examples. Afterwards Chapter 5 will give an introduction and overview of historical and

[1] https://github.com/gleiss/rapid

current state-of-the-art research in software verification relevant to this work and finally conclude in Chapter 6.

CHAPTER 2

Preliminaries

2.1 First-order Logic with Theories

We consider standard many-sorted first-order logic with built-in equality denoted by $\simeq$, that is $\simeq$ is not a symbol, modulo background theories. Besides standard boolean connectives as well as existential and universal quantification, we allow to express inequalities and write $s \not\simeq t$ instead of $\neg(s \simeq t)$ for arbitrary terms s and t. For a valid formula of the form $F_1 \wedge \ldots \wedge F_n \rightarrow F$, we write $F_1, \ldots, F_n \vDash F$. In particular, we say $\vDash F$, if F is valid.

A *signature* Σ is any finite set of symbols. The signature of a formula F is the set of all symbols occurring in F. Let $F := \forall x \,.\, c \simeq f(x)$, then the signature of F is the set $\{f, c\}$. A first-order background *theory* T is a set of all logical consequences of the *theory axioms* of T, i.e. a set of all valid formulas on a class of first-order structures. To this end, when making use of a theory T we call symbols in the signature Σ_T of T *interpreted*, and all other symbols *uninterpreted*.

Specifically, we make use theory of linear arithmetic $T_{\mathbb{I}}$ to reason over integers denoted by sort $\mathbb{I}$, as well as the theory of term algebras T_A [KRV17] to encode loop iterations as natural numbers denoted by sort $\mathbb{N}$ as discussed in 2.3.

The signature of natural numbers $\Sigma_{\mathbb{N}}$ is the set of standard symbols $\{\mathtt{0}, \mathtt{succ}, \mathtt{pred}, <\}$ interpreted as *zero*, *successor*, *predecessor* and *less* respectively. Note that the theory of term algebras comes equipped with the symbols $\{\mathtt{0}, \mathtt{succ}, \mathtt{pred}\}$ which we extended with a proper less-symbol $<$ incompletely axiomatized to have an ordering on natural numbers $\mathbb{N}$.

Theory Reasoning in the presence of full first-order quantification still tends to be a challenge for automated reasoners, thus we could observe that differentiations in theory encodings lead to different results. For example, while we could also encode loop iterations by integers using sort $\mathbb{I}$, due to a blow-up in theory axioms of linear arithmetic, encoding iterations as $\mathbb{N}$ has been shown to dramatically increase performance and the number of problems we could solve. Thus, as opposed to $\mathbb{I}$, $\mathbb{N}$ does not contain interpreted symbols for arbitrary addition and multiplication. The axiomatization of $<$ is incomplete and only contains two axioms.

The signature of integers $\Sigma_{\mathbb{I}}$ is the set of integer constants $0, 1, 2, \ldots$ and a set of operations given by $\{+, *, <\}$ to denote the function symbols for *addition* and *multiplication* as well as the predicate symbol for *less* respectively. Sort $\mathbb{I}$ is used to represent and reason about program variables such as integers or integer-valued arrays (see Section 2.3). We use incomplete but sound axiomatizations supported by the built-in theory of linear arithmetic in VAMPIRE (with setting VAMPIREoptions `-tha` to `some` or `on` for partial, that is considering mostly the axiomatization of the less-symbol, or full axiomatization of $\mathbb{I}$). As we will see in later sections 3 and 4, we extend the built-in partial axiomatization provided by `-tha some` with specific integer theory axioms needed.

Additionally, we consider two uninterpreted sorts to denote timepoints, that is sort (i) Timepoint denoted by $\mathbb{L}$ and sort (ii) Trace denoted as $\mathbb{T}$ used to refer to computation traces which we use for relational verification (see Chapter 4).

Given a variable of first-order logic x and a sort S, we denote x is of sort S as x^S. We use standard first-order models modulo a background theory T. We write $\vDash_T F$ to denote that F is T-valid, that is F holds in all models of T. If I is a model of T, we write $I \vDash_T F$ if F holds in the interpretation I.

2.2 Input Language $\mathcal{W}$

As input to the RAPID framework, we consider programs written in a while-like programming language, denoted as $\mathcal{W}$. $\mathcal{W}$ allows us to express mutable and immutable, that is constant, integer variables and integer-valued arrays. $\mathcal{W}$ includes side-effect free expressions over booleans and integers. Each program consists of a top-level function `main` without arguments and allows for arbitrary nestings of program variable assignments and control-flow structures such as skip , if-then-else and while-statements. Hence, when we refer to loops we speak of while-loops. For all statement `s` that occur in a loop, we refer to these loops as *enclosing loops* of `s`. The semantics of $\mathcal{W}$ will be given in the next section.

2.3 Trace Logic

We now introduce the concept of *trace logic* for expressing semantics and (relational) properties of $\mathcal{W}$-programs as discussed in our recent work [BEG+19].

2.3.1 Syntax

Locations and Timepoints

Programs in $\mathcal{W}$ are considered as sets of program locations, where each location intuitively corresponds to a step in the program execution, that is for each program statement `s`, we introduce a symbol l_s to denote the location. We use l_{end} to denote the location corresponding to the end of the program.

Since we consider programs with arbitrary nesting of loops, locations can be visited multiple times throughout program execution. Thus, we model program locations as follows: (i) for each

location l_s corresponding to a program statement $\mathtt{s}$, we introduce a function symbol $l_s : \mathbb{N}^n \mapsto \mathbb{L}$, that is l_s is of sort $\mathbb{L}$.
(ii) for each enclosing loop of statement $\mathtt{s}$, l_s gets an argument of sort $\mathbb{N}$ that represents the loop iteration of each enclosing loop respectively.
(iii) for while-statements, we additionally equip the encoding with a function symbol $n_s : \mathbb{N}^n \mapsto \mathbb{N}$ of sort $\mathbb{N}$ to denote the iteration where $\mathtt{s}$ terminates, i.e. the first iteration where the loop condition of $\mathtt{s}$ does not hold anymore. Enclosing loops are handled as above as arguments to n_s.

Further we introduce some terms to denote the most commonly used timepoints when discussing semantics. Let it^s be a function that returns a unique variable of sort $\mathbb{N}$ for each while-statement $\mathtt{s}$. Let $w_1, \ldots, w_k$ denote the enclosing loops of some statement $\mathtt{s}$ and let it be an arbitrary term of sort $\mathbb{N}$.
We define the following macros to denote specific timepoints of the program execution:

$$\begin{aligned} tp_{\mathtt{s}} &:= l_s(it^{w_1}, \ldots, it^{w_k}) && \text{if } \mathtt{s} \text{ is not while-statement} \\ tp_{\mathtt{s}}(it) &:= l_s(it^{w_1}, \ldots, it^{w_k}, it) && \text{if } \mathtt{s} \text{ is while-statement} \\ lastIt_{\mathtt{s}} &:= n_s(it^{w_1}, \ldots, it^{w_k}) && \text{if } \mathtt{s} \text{ is while-statement} \end{aligned}$$

Further for an arbitrary program statement $\mathtt{s}$, we define the start of its execution as

$$start_{\mathtt{s}} := \begin{cases} tp_{\mathtt{s}}(\mathtt{0}) & \text{if } \mathtt{s} \text{ is while-statement} \\ tp_{\mathtt{s}} & \text{otherwise} \end{cases}$$

and the end of its execution as

$$end_{\mathtt{s}} := \begin{cases} start_{\mathtt{s}'} & \text{if } \mathtt{s}' \text{ occurs after } \mathtt{s} \text{ in a context} \\ end_{\mathtt{s}'} & \text{if } \mathtt{s} \text{ is last statement in if-branch of } \mathtt{s}' \\ end_{\mathtt{s}'} & \text{if } \mathtt{s} \text{ is last statement in else-branch of } \mathtt{s}' \\ tp_{\mathtt{w}}(\mathtt{succ}(it^w)) & \text{if } \mathtt{s} \text{ is last statement in body of } \mathtt{w} \end{cases}$$

where $end_{\mathtt{s}}$ denotes the first timepoint after the completed execution of statement $\mathtt{s}$.

Program Variables and Expressions

Our reasoning tasks involve reasoning about (arbitrary) values of program variables. To this end, we model program variables $\mathtt{v}$ as functions over timepoints of sort $\mathbb{L}$ and for relational verification also over execution traces of sort $\mathbb{T}$. Precisely, for mutable variables we obtain the following functions

(i) $v : \mathbb{L} \mapsto \mathbb{I}$ for integer-valued program variables,
(ii) $v : (\mathbb{I} \times \mathbb{L}) \mapsto \mathbb{I}$ for integer-valued array variables in the non-relational case,
(iii) $v : (\mathbb{L} \times \mathbb{T}) \mapsto \mathbb{I}$ for integer-valued program variables and
(iv) $v : (\mathbb{I} \times \mathbb{L} \times \mathbb{T}) \mapsto \mathbb{I}$ for integer-valued array variables in the relational case.

Thus $v(tp)$ and $v(tp, tr)$ denote the value of program variable v at timepoint tp and at computation trace tr for the latter. Note that for immutable program variables, we omit the timepoint as an argument to the function. That is the value of a constant variable $\mathtt{v}$ remains unchanged throughout computation is simply denoted as v or $v(tr)$ in the relational case as constants might differ depending on the computation trace.

We now consider arbitrary program expressions $\mathtt{e}$. We write $[\![\mathtt{e}]\!](tp)$ and $[\![\mathtt{e}]\!](tp, tr)$ to denote the value of $\mathtt{e}$ at timepoint tp, in trace tr for the latter. Further, we introduce two definitions expressing properties about values of expressions $\mathtt{e}$ at arbitrary timepoints and traces for the relational setting. Consider now $v \in S_V$, where S_V is the set of function symbols denoting program variables, and let tp_1, tp_2 be two arbitrary timepoints. We define:

$$Eq(v, tp_1, tp_2) := \begin{cases} \forall pos_{\mathbb{I}}.\ v(pos, tp_1, tr) \simeq v(pos, tp_2, tr), & \text{if } \mathtt{v} \text{ is array} \\ v(tp_1, tr) \simeq v(tp_2, tr), & \text{otherwise} \end{cases} \tag{2.1}$$

That is, $Eq(v, tp_1, tp_2)$ in (2.1) states that the program variable $\mathtt{v}$ has the same values at tp_1 and tp_2. We also define:

$$\mathit{EqAll}(tp_1, tp_2) := \bigwedge_{v \in S_V} Eq(v, tp_1, tp_2), \tag{2.2}$$

asserting that all program variables have the same values at the timepoints tp_1 and tp_2.

Note that, the definitions refer to the relational case, that is the functions of program variables take a trace argument tr. For the non-relational setting, we use the same definitions by omitting the trace argument tr for program variables.

2.3.2 Semantics of $\mathcal{W}$

Consider an arbitrary but fixed program P in $\mathcal{W}$. We express semantics of $\mathcal{W}$ in *trace logic* $\mathcal{L}$, by firstly stating *trace axioms of* $\mathcal{L}$ that capture behavior of programs relative to P.

Note that for simplicity we state semantics in the relational setting, that is program variables take a trace argument tr. As above, all definitions still hold for the non-relational case by omitting the trace argument tr from all definitions.

Main-function Let $\mathtt{s}_1, \ldots, \mathtt{s}_\mathtt{k}$ be statements and P be a program with top-level function `func main {`$\mathtt{s}_1$`;...;`$\mathtt{s}_k$`}`. We define the semantics of P as the conjunction of the semantics of each statement $\mathtt{s}_\mathtt{i}$, where $0 \leq i \leq k$. That is:

$$[\![P]\!] := \bigwedge_{i=1}^{k} [\![\mathtt{s}_\mathtt{i}]\!]. \tag{2.3}$$

Thus, the semantics of P are given by structural induction over each program statement $\mathtt{s}$ defined as follows.

Skip Let $\mathtt{s}$ be a `skip`-statement. The evaluation of $\mathtt{s}$ has no effect on the value of the program variables. Hence:

$$[\![\mathtt{s}]\!] := \bigwedge_{v \in S_V} Eq(v, end_{\mathtt{s}}, tp_{\mathtt{s}}) \tag{2.4}$$

Integer assignments Let s be an assignment

```
v = e
```

where v is an integer program variable and e is an expression. The assignment s is evaluated in one step. After its evaluation, variable v has the same value as e before the evaluation. All other variables remain unchanged. We have,

$$[\![\mathtt{s}]\!] := v(end_{\mathtt{s}}) \simeq [\![\mathtt{e}]\!](tp_{\mathtt{s}}, tr) \wedge \bigwedge_{v' \in S_V \setminus \{v\}} Eq(v', end_{\mathtt{s}}, tp_{\mathtt{s}}) \tag{2.5}$$

Array assignments Let s be an assignment

```
a[e1] = e2
```

,

where a is an array variable and $\mathtt{e}_1, \mathtt{e}_2$ are expressions. As above, assignments are evaluated in one step, such that after the evaluation a at position *pos*, corresponding to the value of $\mathtt{e}_1$ before evaluation, corresponds to the value of $\mathtt{e}_2$ before the evaluation of s (2.6a). All other positions of a and other program variables remain unchanged as defined in 2.6b and 2.6c respectively. Hence,

$$\begin{aligned}
[\![\mathtt{s}]\!] := \quad & a(end_{\mathtt{s}}, e_1(tp_{\mathtt{s}}, tr)) \simeq e_2(tp_{\mathtt{s}}, tr) && \text{(2.6a)}\\
\wedge \quad & \forall pos_{\mathbb{I}}.(pos \not\simeq e_1(tp_{\mathtt{s}}, tr) \rightarrow \\
& a(end_{\mathtt{s}}, pos, tr) \simeq a(tp_{\mathtt{s}}, pos, tr)) && \text{(2.6b)}\\
\wedge \quad & \bigwedge_{v \in S_V \setminus \{a\}} Eq(v, end_{\mathtt{s}}, tp_{\mathtt{s}}) && \text{(2.6c)}
\end{aligned}$$

Conditional if-then-else Statements Let s be the statement

```
if(Cond){s1;...;sk} else {s'1;...;s'k'}.
```

The semantics of s is defined by the following two properties: (i) entering the if- or else-branch does not change the values of variables as defined in 2.7a, (ii) the evaluation in each branch that is entered is defined inductively according to the semantics of its respective statements (2.7b) Thus:

$$\begin{aligned}
[\![\mathtt{s}]\!] := \quad & [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}) \rightarrow EqAll(start_{\mathtt{s}_1}, tp_{\mathtt{s}}) \\
\wedge \quad & \neg[\![\mathtt{Cond}]\!](tp_{\mathtt{s}}) \rightarrow EqAll(start'_{\mathtt{s}_{1'}}, tp_{\mathtt{s}}) && \text{(2.7a)}\\
\wedge \quad & [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}) \rightarrow [\![\mathtt{s}_1]\!] \wedge \cdots \wedge [\![\mathtt{s}_k]\!] \\
\wedge \quad & \neg[\![\mathtt{Cond}]\!](tp_{\mathtt{s}}) \rightarrow [\![\mathtt{s'}_1]\!] \wedge \cdots \wedge [\![\mathtt{s'}_{k'}]\!] && \text{(2.7b)}
\end{aligned}$$

While-Loops Let s be the while-statement

```
while(Cond){s1;...;sk}.
```

We refer to Cond as the *loop condition*. The semantics of s is defined in terms of the following properties: (i) the iteration $lastIt_{\mathtt{s}}$ denotes the first iteration where the loop condition does not hold (2.8a), (ii) entering the loop body does not change variable values (2.8b), (iii) the evaluation in the body proceeds according to the semantics of the statements in the body (2.8c), (iv) the

variable values after the evaluation of $\mathtt{s}$ coincide with the values in iteration $lastIt(\mathtt{s})$ at the location of the beginning of the loop, that is the location of condition $\mathtt{Cond}$ (2.8d). We then have:

$$
\begin{aligned}
[\![\mathtt{s}]\!] := \quad & \forall it^s_{\mathbb{N}}.\ (it^s < lastIt_{\mathtt{s}} \rightarrow [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}(it^s))) \\
\wedge \quad & \neg[\![\mathtt{Cond}]\!](tp(lastIt_{\mathtt{s}})) && \text{(2.8a)} \\
\wedge \quad & \forall it^s_{\mathbb{N}}.\ (it^s < lastIt_{\mathtt{s}} \rightarrow EqAll(start_{\mathtt{s}_1}, tp_{\mathtt{s}}(it^s)) && \text{(2.8b)} \\
\wedge \quad & \forall it^s_{\mathbb{N}}.\ (it^s < lastIt_{\mathtt{s}} \rightarrow ([\![\mathtt{s}_1]\!] \wedge \cdots \wedge [\![\mathtt{s}_k]\!]) && \text{(2.8c)} \\
\wedge \quad & EqAll(end_{\mathtt{s}}, tp_s(lastIt_{\mathtt{s}})) && \text{(2.8d)}
\end{aligned}
$$

2.3.3 Trace Logic $\mathcal{L}$

We now define *trace logic* $\mathcal{L}$, allowing us to reason about both reachability and relational properties of programs.

For the relational setting, we define S_{Tr} to be a set $\{t_1, t_2, \dots\}$ of nullary function symbols, that is constants, of sort $\mathbb{T}$. Intuitively, these symbols allow us to denote and express properties over multiple traces. The signature of $\mathcal{L}$ contains the symbols of theories $\mathbb{N}$ and $\mathbb{I}$ together with the symbols introduced in Section 2.3.1, notably a set of timepoints S_{Tp}, last iterations in loops denoted by S_n, program variables S_V and traces S_{Tr}. Formally,

$$Sig(\mathcal{L}) \quad := \quad (S_{\mathbb{N}} \cup S_{\mathbb{I}}) \quad \cup \quad (S_{Tp} \cup S_n \cup S_V \cup S_{Tr}).$$

Recall that the semantics of $\mathcal{W}$ is defined by the trace axioms (2.4)-(2.8). By extending standard small-step operational semantics with timepoints and traces, we obtain the small-step semantics of $\mathcal{W}$. Details and proof of soundness can be found in the Appendix of our recent work [BEG+19].

For proving soundness, of this semantics, we rely on so-called *execution-interpretation* of a program execution E: such an interpretation is a model in which for every (array) variable $\mathtt{v}$ the term $v(tp_i)$ resp. $v(tp_i, pos)$ is interpreted as the value of $\mathtt{v}$ at the execution step in E corresponding to timepoint tp_i. We then refer to the soundness of the semantics of $\mathcal{W}$ as $\mathcal{W}$-soundness, defined as:

Definition 1 ($\mathcal{W}$-Soundness)**.** *Let p be a program and let A be a trace logic property. We say that A is* $\mathcal{W}$-sound*, if for any execution-interpretation M we have $M \vDash A$.*

By using structural induction over program statements, we derive $\mathcal{W}$-soundness of the semantics of $\mathcal{W}$. That is:

Theorem 1 ($\mathcal{W}$-Soundness of Semantics of $\mathcal{W}$)**.** *For a given terminating program p, the trace axioms (2.4)-(2.8) are $\mathcal{W}$-sound.*

As a consequence, the semantics of any terminating program P expressed in $\mathcal{L}$, as defined in (2.3), is $\mathcal{W}$-sound.

2.3.4 Program Correctness in Trace Logic $\mathcal{L}$

Let P be a program and F be a first-order property of P, with F expressed in $\mathcal{L}$. We use $\mathcal{L}$ to express and prove that P "satisfies" F, that is P is partially correct w.r.t. F, as follows:

1. We express $[\![P]\!]$ in $\mathcal{L}$, as discussed in Section 2.3.2;
2. We prove the partial correctness of P with respect to F; that is, we prove

$$[\![P]\!] \vDash_{\mathbb{NUI}} F.$$

2.3.5 Trace Lemmas

In what follows, we first discuss proving (non-relational) reachability properties F over programs expressed in $\mathcal{L}$ (Chapter 3) and then focus on proving partial correctness for relational problems using $\mathcal{L}$ (Chapter 4).

To this end, we defined different sets of *trace lemmas* guiding the prover in inductive reasoning steps to automatically perform proofs over programs containing (nested) loops. Trace lemmas are statically inferred from the program semantics. They express inductive properties about the program behavior. We will illustrate how to use trace lemma reasoning for different sets of programs and properties in nonrelational and relational settings.

2.3.6 Experiments and Tooling

To generate program semantics as described in the previous sections we rely on the RAPID framework, where we also implemented this work, that is the automatic instantiation of trace lemmas. The proofs of the problems in this work rely on the first-order theorem prover VAMPIRE [KV13] based on the superposition calculus, notably a refutational full first-order prover with built-in equality reasoning and theory support.

CHAPTER 3

Non-relational Trace Lemma Reasoning

In this chapter, we describe our approach to using first-order reasoning in trace logic $\mathcal{L}$ for automating program analysis and verification. We illustrate our work using challenging examples from the verification repository of the annual *Competition on Software Verification (SV-COMP)*[1].

First, we will highlight some standard array reasoning examples and explain how our approach is applied to two examples in detail. Second, we will showcase the power of our approach by proving seemingly simple examples that simulate the break of a loop - a problem whose proofs are still not completely automated. Finally, we showcase the scalability of the approach by emphasizing a proof of a property with a quantifier alternation for a larger program with nested loops. Note that, while the translation of the program semantics to first-order logic generates many more trace lemmas, we will focus on the ones used by the refutational prover to find the empty clause for each example individually to give a proof-of-concept of our approach and show how proofs are performed.

3.1 Array Reasoning

We first discuss our approach for proving reachability properties, by illustrating our work on examples involving reasoning over arrays.

3.1.1 Searching and finding elements in arrays

We consider programs as in, or similar to, Figure 3.1: such a program traverses an integer array `a` until it finds the array element corresponding to the value `v`. We want to show that given the array `a` is non-empty and the value of loop counter variable `i` is smaller than the length of array

[1] https://sv-comp.sosy-lab.org/

```
 1    func main()
 2    {
 3      const Int[] a;
 4      const Int alength;
 5      const Int v;
 6      Int i = 0;
 7
 8      while (i < alength && a[i] != v)
 9      {
10        i = i + 1;
11      }
12    }
13
```

Figure 3.1: Find an element v.

a after the computation, then we know that there exists a position pos in the array, such that the value of a$[pos]$ is equal to v. Formally we obtain property 3.1:

$$alength \geq 0 \rightarrow \exists pos^{\mathbb{I}} \,.\, alength > i(end) \rightarrow a(pos) = v \tag{3.1}$$

where $i(end)$ encodes the value of the loop counter i after the program execution and $pos_{\mathbb{I}}$ denotes that pos is of sort integer.

Essentially, we can prove this example from the customized program semantics in trace logic that encode with every non-constant program variable v used in the loop a function $v : \mathbb{L} \rightarrow \mathbb{I}$ over program locations l and loop iterations it of sort $\mathbb{N}$, thus allowing us to refer to iteration-specific values of v.

Specifically, program semantics in trace logic allow us to infer that if $\texttt{i}(l_8(end)) < \texttt{alength}$ holds, where l_8 denotes the program location of the while-loop, we can infer that a at the position defined of $i(l_8(end))$ is equal to the sought value v, i.e. we obtain clause (1) $\texttt{i}(l_8(end)) < \texttt{alength} \rightarrow \texttt{a}(i(l_8(end))) = \texttt{v}$. This actually follows from the loop condition stating that $\texttt{i} < \texttt{alength} \wedge \texttt{a(i)} \neq \texttt{v}$. At the end of the loop's execution, i.e. at timepoint end, we know that in case $\texttt{i} < \texttt{alength}$ still holds, $\texttt{a(i)} \neq \texttt{v}$ cannot hold at the same time, as this would imply another loop iteration.

Now from the negation and clausification of the property, we obtain two clauses stating that (2) $\texttt{a}(x) \neq v)$ where x is a variable implicitly universally quantified, i.e. for any position x the clause holds during saturation, and (3) $i(end) < alength$.

Now one can quickly see that unifying (1) and (2) is subsumed by a clause of the form $\texttt{i}(l_8(end)) < \texttt{alength} \rightarrow false$ which in combination with (3) is subsumed by $false$. Hence we obtain the empty clause, and prove that the property is correct for the given program.

3.1.2 Initializing arrays

Figure 3.2 shows an example for array initialization: given an array a of length alength, the program initializes the array with some integer value v. The property we want to prove is that at every position of a the stored value is v after the execution. More formally

```
 1    func main()
 2    {
 3      Int[] a;
 4      const Int alength;
 5      const Int v;
 6      Int i = 0;
 7
 8      while(i < alength)
 9      {
10        a[i] = v;
11        i = i+1;
12      }
13    }
14
```

Figure 3.2: Array initialization.

$$\forall pos^{\mathbb{I}} \,.\, (0 \leq pos < alength \wedge 0 \leq alength) \rightarrow a(end, pos) = v \tag{3.2}$$

where end denotes the last time point after the execution.

Intuitively, it is clear why proving this problem should succeed: (i) we know that due to the incrementation of `i` from 0 to $(\texttt{alength} - 1)$, every position of `a` in this range will be affected by exactly one of the loop iterations. (ii) We also know that once `v` is assigned to a position in the array, it will not be changed in the future. For proving (i), we need to ensure that `i` will have the value of every integer in its range at least once, which is formalized by the following trace lemma:

$$\begin{aligned} &\forall it_8^{\mathbb{N}}, x^{\mathbb{I}} \,. \\ &\quad \Big(i(l_8(it_8(zero))) \leq x < i(l_8(it_8(lastIt_8))) \\ &\quad \wedge\, i(l_8(s(it_8)) = i(l_8(it_8)) + 1 \Big) \\ &\quad \rightarrow \exists it^{\mathbb{N}} \,.\, i(l_8(it)) = x \wedge it < lastIt_8 \end{aligned} \tag{3.3}$$

where l_8 (`line 8`) denotes the program location of the loop. Lemma (3.3) states that for every integer x smaller than the last iteration of the loop, there exists an iteration where the value of the loop counter `i` is equal to the value x.

To reason about (ii), we use a trace lemma similar to (3.3), intuitively asserting that an element of `a` changed at a loop iteration `i` will not be changed in further iterations, i.e. that the value of `a` at some position once assigned remains unchanged throughout computation. Particularly, we need to express that if we know that `a` at some position is assigned `v` within some iteration between 0 and $(alength - 1)$, it will not be changed by any succeeding iteration of the loop. To generalize this idea, we can deduce that for every position of `a` within these bounds, also for any potentially larger left bound of the assignment and smaller right bound in this range, once assigned, the value remains unchanged in all iterations it up to the right bound. We formalize lemma (3.4):

$$\begin{aligned}&\forall boundL^{\mathbb{N}}, boundR^{\mathbb{N}}, pos^{\mathbb{I}}.\\&\quad\Big(\forall it^{\mathbb{N}}\,.\,(boundL \leq it < boundR \wedge a(l_8(boundL), pos) = a(l_8(it), pos))\\&\qquad\rightarrow a(l_8(boundL), pos) = a(l_8(s(it)), pos)\Big)\\&\rightarrow a(l_8(boundL), pos) = a(l_8(boundR), pos)\end{aligned} \tag{3.4}$$

This inductive lemma states that for any left and right bounds, if the value of `a` is the same at each position within these bounds, then it is particularly also equal from the left to right bound. With this lemma, we ensure the inductive step over the loop: once the value at some position in the array is set, it is not changed after. With such trace lemmas about (i) and (ii), our superpostion-based reasoning in trace logic automatically proves (3.2).

3.2 Simulating Break Statements

3.2.1 Break decrementing

```
1     func main()
2     {
3       Int x;
4       Int found = 0;
5
6       while(found == 0)
7       {
8         if(x > 0)
9         {
10          x = x - 1;
11        }
12        else
13        {
14          found = 1;
15        }
16      }
17    }
18
```

Figure 3.3: Break when $x = 0$.

To illustrate the advantages of using explicit timepoints in the encoding of program semantics in trace logic, we consider the following two examples that simulate the functionality of a `break` statement. In most imperative programming languages, the break command is a control statement to preemptively terminate loops. Once break is executed within a loop, we immediately jump to the next statement after the current loop - usually to terminate loops based on some condition.

To this end, the program in Figure 3.3 simulates this behavior by terminating the loop once the program variable `found` is set to 1. Given the conditional in the loop, we know that `found`

will have the value 1 once x is actually equal to or smaller than 0. Hence, this is also the property we want to check for this program:

$$x(end) \leq 0 \tag{3.5}$$

Intuitively, the proof is very easy for humans to understand: (i) either x is already smaller than or equal to 0 at the beginning of the program execution which means that there is only one loop iteration immediately setting the variable found to 1 and terminating the program execution. Property 3.5 is thus trivially satisfied. (ii) On the other hand, if x is greater than 0, the loop will actually be executed due to the assignment right before the beginning of the loop making the loop condition true. From this and the fact that we prove partial correctness, that is we take termination for granted, we can infer that the if-statement will be executed until x is 0.

Similar to this reasoning, we need to make sure that the prover is equipped with the information that there must be at least on loop iteration, which we formalize as the following lemma:

Let C be the expression that is the loop condition, then we know that there exists a loop iteration it that is smaller than $lastIt$ which refers to the timepoint, i.e. the first iteration, where the loop condition does not hold anymore.

$$C \rightarrow \exists it^{\mathbb{N}} \centerdot s(it) = lastIt \tag{3.6}$$

Hence, the instantiated lemma for program 3.3 looks like the following:

$$found(l_6(zero)) = 0 \rightarrow \exists it^{\mathbb{N}} \centerdot s(it) = lastIt_6 \tag{3.7}$$

Now, together with the fact that every timepoint is smaller than its successor, which is formalized by the following axiom

$$\forall it^{\mathbb{N}} \centerdot it < s(it) \tag{3.8}$$

we can actually infer that there is an iteration $it < lastIt_6$, thus allowing the prover to deduce that the loop is executed. Now the particularity of this proof is that all further reasoning over the value of x can be deduced merely from the program semantics and the axiomatization of the less-symbol for integers. Informally this works as follows: By knowing that there is an it that is exactly one step away from $lastIt_6$, the prover knows that this is the iteration where found is set to 1. Now from the conditional in the loop, we know that the if-part is not executed, thus $x > 0$ is not true anymore. Hence $x(end) \leq 0$ is true. Now, for the refutational proof, the prover negates the property, such that we also have the clause $x(end) > 0$ in the search space. Now from the axiomatization of the less-symbol over the integers, particularly from the antisymmetry axiom

$$\forall x_0^{\mathbb{I}} \centerdot x_0 < x_0 \rightarrow false \tag{3.9}$$

the prover can derive the empty clause from the fact that $x(end) \leq 0 < x(end)$ clearly cannot hold.

```
 1    func main()
 2    {
 3      const Int k;
 4      Int x;
 5      Int found = 0;
 6
 7      while(found == 0)
 8      {
 9        if(x < k)
10        {
11          x = x + 1;
12        }
13        else
14        {
15          found = 1;
16        }
17      }
18    }
19
```

Figure 3.4: Break at k.

3.2.2 Break incrementing

A reasoning similar as in section3.2.1 can also be applied to the example in Figure 3.4 which is another variation of a break simulation that simulates finding the k-th element and breaking once it was found. The property is thus adjusted to the following:

$$x(end) \geq k \tag{3.10}$$

As in section3.2.1, the prover knows that there must be at least one iteration, hence we know that at the last iteration of the loop, the conditional within the loop does not hold anymore which translates to a clause of the $x(end) \geq k$ in the search space. Together with the negated property $x(end) < k$, we immediately can derive the empty clause without needing any further loop invariants that handle induction over the values of x. This clearly indicates the potential power of reasoning over timepoints as other automated provers such as Dafny still require users to provide the right loop invariants to automatically prove this property.

3.3 Quantifier Alternation

Consider the example in Figure 3.5: the program simulates the comparison of two arrays. In our case we want to verify that the values of `signatures` and `storedSignatures` coincide. For every stored signature, we compare the array values with a nested loop construct and set `foundMissing` to 1 in case there exists a position in the `storedSignatures` array such that there is no value in `signatures` corresponding to the value at this position of `storedSignatures`. Given this specification it is natural to verify a property that states that

```
1   func main()
2   {
3     const Int[] storedSignatures;
4     const Int[] signatures;
5     const Int storedSignaturesLength;
6     const Int signaturesLength;
7
8     Int foundMissing = 0;
9     Int i = 0;
10
11    while (i < storedSignaturesLength)
12    {
13      Int found = 0;
14      Int j = 0;
15      while (j < signaturesLength)
16      {
17        if (storedSignatures[i] == signatures[j])
18        {
19          found = 1;
20        }
21        else
22        {
23          skip;
24        }
25        j = j + 1;
26      }
27
28    if(found == 0)
29    {
30      foundMissing = 1;
31    }
32    else
33    {
34      skip;
35    }
36
37    i = i + 1;
38    }
39  }
40
```

Figure 3.5: Find missing signature.

in the case of a differentiation in the input arrays `storedSignatures` and `signatures`, the variable `foundMissing` will be set to 1 at the end of the computation. Formally we obtain the following property:

$$\begin{aligned}&\Big(\exists pos^{\mathbb{I}} \,.\, 0 \leq pos < storedSignaturesLength \wedge \\ &\quad \forall i^{\mathbb{I}} \,.\, storedSignatures(pos) \neq signatures(i)\Big) \\ &\rightarrow foundMissing(end) = 1\end{aligned} \tag{3.11}$$

As we can see, we need a quantifier alternation to specify this behavior: there exists a position in the `storedSignatures` array such that no value stored in `signatures` is equal.

This example highlights two major strengths of our approach:

- (1) *Arbitrary Loop Nesting.* By having explicit time points of loops, we can always extend the sort of any timepoint of and within nested loops with the iteration of all outer loops as well. Here, this means that time point l_{15} is a function over two natural numbers indicating the iterations of the outer and inner loop respectively: $l_{15} : \mathbb{N}^2 \rightarrow \mathbb{L}$. With this encoding all of our trace lemmas can be instantiated in the same way as before, thus still hold for nested loops. However as we will see below, some specific reasoning is needed with nested loops.

- (2) *Quantifier alternations.* By using the full first-order theorem prover VAMPIRE, we can handle statements like property 3.11 consisting of a $\exists\forall$ alternation.

Intuitively, the proof works in the following way: Given that the variable in question `foundMissing` is 0 in the beginning of the execution and 1 after the computation, there must be some timepoint where the variable is updated. Since, this is exactly the case when the execution of the inner loop does not find the appropriate match when comparing array values, we know that this timepoint must exist. Further, if we can establish that if the variable is set to 1 and is not updated after, we know that `foundMissing` is also 1 at the end. We can also easily infer this, since there is no other update within the loops that would change the value of the variable. To guide the prover in establishing this, we need to consider trace lemmas as follows.

The first lemma that needs to be instantiated in order to relate iterations of sort $\mathbb{N}$ and array positions of sort $\mathbb{I}$ is the so-called *intermediate value lemma* for the loop counter variable `i`:

$$\begin{aligned}&\forall it_{11}^{\mathbb{N}}, it_{15}^{\mathbb{N}}, x^{\mathbb{I}}. \\ &\quad \Big(i(l_{15}(it_{11}(zero))) \leq x < i(l_{15}(it_{11}(lastIt_{15}(it_{11})))) \\ &\quad \wedge i(l_{15}(it_{11}(s(it_{15})))) = i(l_{15}(it_{11}(it_{15}))) + 1\Big) \\ &\quad \rightarrow \exists it^{\mathbb{N}} \,.\, i(l_{15}(it_{11}(it))) = x \wedge it < lastIt_{15}(it_{11})\end{aligned} \tag{3.12}$$

where $lastIt_{15}(it_{11})$ denotes the last iteration of the inner loop in iteration it_{11} of the outer loop. We have seen in this lemma in previous sections. However, in this particular case one can see a major advantage of the explicit form of using timepoints in the encoding: we can explicitly refer to the last iteration of the inner loop while also explicitly knowing the iteration of the outer loop. This modularity technically allows us to nest loops arbitrarily.

Further, we need a way to establish that the value of `found` will be 0 after the execution of the while loop. For this we need induction over the equality of the value of `found` in all iterations of the inner loop, which is expressed in the following way:

$$\begin{aligned}&\forall it_{11}^{\mathbb{N}}.\\&\quad\Big(\forall it_{15}^{\mathbb{N}}\,.\,it_{15} < lastIt(it_{11}) \rightarrow found(l_{15}(it_{11}(it_{15}))) = found(l_{15}(it_{11}(s(it_{15}))))\Big)\\&\quad\rightarrow found(l_{15}(it(zero))) = found(l_{15}(it_{11}(lastIt(it_{15}))))\end{aligned} \tag{3.13}$$

where it_{11} is the iteration of the outer loop at `line 11` and it_{15} represents the iterations of the inner loop at `line 15`. This lemma comes into play exactly at the execution of the inner loop, where `found` will never be 1, that is where the array `signatures` does not contain any match for the current value of `storedSignatures` at position i. This lemma is key in finding that there actually is an update of `foundMissing` after all.

Now that we know of the existence of an update, we need to consider the second part of the reasoning, that is reasoning over the *preservation* of the value 1 for the program variable `foundMissing`. To do so, we instantiate the following lemma expressing *value preservation* over multiple iterations of the outer loop at position l_{11}. Lemma 3.14 informally states that once the variable is assigned to any value x, and it is not changed throughout further iterations, then it will in particular have this value at the end of loop.

$$\begin{aligned}&\forall x^{\mathbb{I}}.\\&\quad\Big(\exists it^{\mathbb{N}}\,.\,it < lastIt_{11} \wedge foundMissing(l_{11}(s(it))) = x\\&\qquad\wedge (\forall it_{11}^{\mathbb{N}}\,.\,(it < it_{11} \wedge foundMissing(l_{11}(it_{11})) = x)\\&\qquad\rightarrow foundMissing(l_{11}(s(it_{11}))) = x)\Big)\\&\quad\rightarrow foundMissing(l_{11}(lastIt_{11})) = x\end{aligned} \tag{3.14}$$

While we are now equipped with the intuitive reasoning a human would perform, there is one more thing humans do very implicitly when looking at such problems: we automatically disregard the fact that for example variable `i` is never changed in the any iteration of the inner loop. While this knowledge allows us to mentally "unroll" or rather *statically* infer information about the inner and outer loop accordingly, the prover needs a lemma to establish this fact.

To this end, we instantiate so-called *static analysis* lemmas, that are used to exclude behavior/updates of program variables that do not appear in loops. Here, the loop counter `i` of the outer loop will never be changed within the inner loop at `line 15`. Thus, we can statically infer - since these variables simply do not appear in the inner loop body - that certain program variables are not influenced by any behavior in the loop, that is they stay the same over all iterations. The following lemma is thus instantiated for both program variables `i` and `foundMissing` with the location of the inner loop l_{15}.

$$\forall it_{11}^{\mathbb{N}}, it_{15}^{\mathbb{N}}\,.\,i(l_{15}(it_{11}(zero))) = i(l_{15}(it_{11}(it_{15}))) \tag{3.15}$$

Finally, with this line of reasoning the prover is able to establish that in fact `foundMissing` will be 1 at the end of the computation under the given circumstances defined in the property. While this is clearly a toy example, the potential of automating this kind of reasoning together with relational verification might have interesting implications for the automation of proving security properties for protocol verifications.

CHAPTER 4

Relational Trace Lemma Reasoning

In the following we overview our work in the relational setting. As in Chapter 3, we illustrate our work on multiple examples categorized by the respective program property, namely noninterference and sensitivity. These properties fall into the category of so-called k-safety properties, in our case 2-safety properties, that is properties that are expressed over two sets of traces. Intuitively, this means these properties might be violated by two program runs, for instance in the case of noninterference running two arbitrary traces on the same inputs and computing different results would be such a violation. Note that, while the translation of the program to trace logic instantiates more trace lemmas, we will highlight the ones used by the refutational prover to find the empty clause for each example individually.

4.1 Noninterference

In general, we want programs to be safe from attackers. Particularly we do not want attackers to be able to access and read our sensible data. These kind of concerns are usually treated with access control mechanisms. However, these do not guarantee that there are no loopholes that can be exploited to access potentially sensitive data during program execution. We especially want to exclude the possibility that secure data might flow into publicly accessible data during program execution. To this end, we will discuss information flow policies and show how to ensure confidentiality by proving *noninterference* [GM82] for some exemplary programs written in our input language $\mathcal{W}$.

Intuitively noninterference is the property that prevents secret data from flowing into the public program state, that is accessible for an attacker. In other words, secret and public information do not interfere with each other. Informally, we prove noninterference by ensuring for a program P that if the public input of two arbitrary traces (i.e. the public program state of two program executions) are equal before their respective execution, then the public output at the end will also be equal, thus ensuring that any sensitive data that might differ in the traces does not interfere with the result of the computation that is publicly accessible. Hence, we say that P is noninterfering.

Note that we assume termination of execution traces, thus we generally prove partial correctness with our approach. Precisely, we divide the program state of P into high and low confidentiality variables, denoted by H and L respectively. If the input for all L variables is the same in both runs of P, then the output of L variables should also have the respective equal values in each trace, independently of any values of the variables in H. Formally, we define noninterference in trace logic $\mathcal{L}$ in the following.

Let l_0 denote the timepoint before the program execution and let $EqTr(v, tp)$ denote that variable v has the same value in both traces at timepoint tp, precisely:

$$EqTr(v, tp) := \begin{cases} \forall pos^{\mathbb{I}}.v(tp, pos, t_1) \simeq v(tp, pos, t_2)) & \text{if } v \text{ is a mutable array} \\ \forall pos^{\mathbb{I}}.v(pos, t_1) \simeq v(pos, t_2)) & \text{if } v \text{ is a constant array} \\ v(tp, t_1) \simeq v(tp, t_2)) & \text{if } v \text{ is a mutable variable} \\ v(t_1) \simeq v(t_2) & \text{if } v \text{ is a constant variable} \end{cases}$$

Then noninterference is formalized as follows:

$$(\bigwedge_{v \in L} EqTr(v, l_0)) \rightarrow (\bigwedge_{v \in L} EqTr(v, l_{end})). \tag{4.1}$$

4.1.1 Explicit flow

```
1   func main()
2   {
3     Int hi;
4     Int lo;
5     Int dec;
6
7     while (dec != 0)
8     {
9       hi = lo + 1;
10      lo = hi + 1;
11      dec = dec - 1;
12    }
13  }
14
```

Figure 4.1: Explicit flow.

Example 1. Consider the program illustrated in figure 4.1. The program contains a so-called explicit flow [SM03], that is we directly store secret data, represented by variable `hi` in a public channel `lo`. Type systems usually forbid such explicit flows as they easily make the program unsafe. However in this case, we can still claim this program to be safe with regards to noninterference as observational equivalence is not violated.

Formally, we prove the following property stating that if all variables in L share the same values in the beginning of the execution, then the values of `lo` are equal after the execution as well:

$$\begin{aligned}&(EqTr(lo, l_0) \land EqTr(dec, l_0) \land \forall tr^{\mathbb{T}} \,.\, (dec, l_0, tr) \geq 0)\\ &\rightarrow EqTr(lo, l_{end})\end{aligned} \tag{4.2}$$

The proof of the example requires the following two trace lemmas that will be discussed below:

$$\begin{aligned}&\Big((\forall it^{\mathbb{N}} \,.\, it < lastIt(t_2) \rightarrow dec(l_7(it), t_1) \neq 0)\\ &\land dec(l_7(lastIt(t_2)), t_1) = 0\Big) \rightarrow lastIt(t_1) = lastIt(t_2)\end{aligned} \tag{4.3}$$

Lemma 4.3 axiomatizes the case that there are equally many iterations in both program executions. Intuitively, if for all iterations smaller than the last iteration in trace t_2, denoted by $lastIt(t_2)$, we have that variable `dec` representing a descending loop counter (of the loop at location l_7) is not equal to 0, that is the loop condition holds, and the value of `dec` in trace t_1 at the last iteration of trace t_2 is 0, we can conclude that the last iteration occurs at the same time in both traces. Hence, the lemma describes how iterations expressed as natural numbers $\mathbb{N}$ in terms of term algebras [KRV17] relate to program variables of VAMPIRE's built-in integer sort $\mathbb{I}$ over multiple traces.

$$\begin{aligned}&\Big(EqTr(v, l_{10}(0)) \land\\ &\forall it^{\mathbb{N}} \,.\, EqTr(v, l_{while}(it)) \rightarrow EqTr(v, l_{while}(s(it)))\Big)\\ &\rightarrow \forall it^{\mathbb{N}} \,.\, EqTr(v, l_{while}(it))\end{aligned} \tag{4.4}$$

Lemma 4.4 expresses induction of the equality of program values over traces and loop iterations, where l_{while} denotes the location of the corresponding while-loop. Note that v merely serves as a placeholder to give the general scheme of Lemma 4.4. We instantiate it for both program variables `lo` and `dec` at the location l_7 of the loop. For both variables, this lemma is needed to reason inductively about the equalities of the values in both traces during the execution of a loop. It states that for some function v representing a program variable, if the values of v are equal in both traces before the loop execution and they are step-wise equal for each iteration during the loop execution - that is their equality is preserved for each following iteration up to the last - then we can infer that the values of v are equal throughout all iterations, particularly also that their values are equal in the last iteration which is essential for the proof.

To give an intuition about proofs of noninterference properties, we give details about how the prover discharges the verification conditions generated by RAPID: Since VAMPIREis a refutational prover, it negates the given property and by conjunctive normal form (CNF) transformation adds the clause $lo(end, t_1) \neq lo(end, t_2)$ to the clause set.

Further the prover establishes that the value of $lo(end)$ is equal to the value of `lo` at the last iteration of the loop in each trace respectively (or rather in all traces, i.e. a clause of the form $lo(end, x) = lo(l_{13}(nl_{10}(x)), x)$ is added to the clause set, where x is implicitly universally

quantified and l_{13} is the location of the assignment to `lo` within the loop and nl_{10} denotes the last iteration of the loop, notably the first iteration where the loop condition does not hold anymore.

Additionally, Lemma4.4 for variable `lo` is used to establish that the values of `lo` are equal in the same iteration in both traces, formally this is expressed by the clause $lo(l_{13}(x), t1) = lo(l_{13}(x), t2)$.

Finally the prover uses Lemma 4.3 to establish that $nl_{10}(t_1) = nl_{10}(t_2)$. In fact, this is also where Lemma4.4 for variable `dec` comes into play as we need to establish the same equalities over traces for `dec` as for `lo` such that lemma 4.3 can effectively be applied.

Now, it is easy to see that we can simply substitute the equalities such that we end up with a clause of the form $lo(end, t_1) = lo(end, t_2)$ and obtain a refutation, hence a proof for the original (non-negated) property.

```
1       func main()
2       {
3         const Int k;
4         const Int lo;
5         Int hi = lo;
6         Int counter = 0;
7         Int[] output;
8
9         while(hi < k){
10          output[counter] = hi;
11          counter = counter + 1;
12          hi = hi + 1;
13        }
14      }
15
16
```

Figure 4.2: Explicit flow with output array.

Example 2. Figure 4.2 illustrates a program outputting on a public channel modeled by the array variable `output` $\in L$. As the naming of variables already implies, `hi` models a secret variable while `lo` $\in L$. Clearly in the loop, we witness an explicit flow at location l_{10} as we directly output the current value of `hi`. Besides, the program also contains a so-called implicit flow, that is an insecure data flow that might give hints about secret variables by the means of the program structure. In our case, using the secret in the guard of the loop, while outputting on a public channel in the body, might reveal information about the secret not only in the content but also in terms of the number of outputs. Indeed, value-insensitive type systems for information flow analysis [SM03] would dismiss this program to be insecure. However, looking closer at the program, the secret value in `hi` is overwritten before the execution of the loop with the L-value stored in `lo`. Thus, the program effectively satisfies noninterference as no secret value stored in `hi` influences any L variables which we can prove thanks to value-sensitivity being inherent to our encoding with $\mathcal{L}$.

Formally, the property we prove is similar to 4.2:

$$(EqTr(k, l_{11}) \wedge EqTr(lo, l_{11}) \wedge EqTr(output, l_{11})) \\ \rightarrow EqTr(output, l_{end}) \tag{4.5}$$

Effectively, this example uses the same lemmas as example 1. Lemma 4.4 is instantiated for variables `counter` and `hi` and in an array-variant for the array variable `output`. Formally for arrays we have to specify the equality over traces over all integers representing the array position as follows:

$$\forall pos^{\mathbb{I}} \,.\, (EqTr(output, pos, l_9(0)) \wedge \\ (\forall it^{\mathbb{N}} \,.\, EqTr(output, pos, l_9(it)) \rightarrow EqTr(output, pos, l_9(s(it)))) \\ \rightarrow \forall it^{\mathbb{N}} \,.\, EqTr(output, pos, l_9(it))) \tag{4.6}$$

The proof mechanism then works as in example 1: Lemma 4.4 for variables `counter` and `hi` and lemma 4.6 handle the inductive reasoning over all variables used in the loop guiding the prover in establishing equal values over traces. Trace lemma 4.3 establishes that the loops in both program executions have equally many iterations. Thus, we can in the same way find a refutation proving noninterference for problem 4.2.

4.1.2 Implicit flow

```
1   func main()
2   {
3     Int hi;
4     Int lo;
5
6     if(hi > 0)
7     {
8       lo = lo + 1;
9     }
10    else
11    {
12      lo = lo + 1;
13    }
14  }
15
```

Figure 4.3: Implicit flow with branching.

Example 3. Figure 4.3 shows an example which branches based on the value of an H variable. While this is considered as a potential implicit flow and therefore rejected by information flow type systems [SM03] as the repeated execution on different inputs might reveal details about which branch is taken, this particular example is safe with regards to noninterference. The problem lies in the fact that the branching decision might leak information about the secret used

in the condition. However, since L variables are updated in the same manner in both branches, the public result of the computation will not differ. Particularly, state-of-the-art static analysis tools based on dependency graphs, such as JOANA [GHM13] fail to prove the program correct with regards to noninterference. Formally, we prove the following property:

$$EqTr(lo, l_0) \rightarrow EqTr(lo, l_{end}). \tag{4.7}$$

While this work is generally about trace reasoning, what makes this example interesting is that the semantics of trace logic allow to prove the example without any trace lemmas as there is no induction needed to prove the program correct. Merely two applications of the superposition rule allow VAMPIREto find a refutation. First the prover finds that the value of `lo` is equal in both traces at `line 12`. (Further, Vampire establishes that the value of `lo` at the end of the computation in trace t_1 is equal to the increment of `lo` at `line 12`. Note that according to semantics $(\texttt{lo}, l_{12})$ denotes the value before the assignment is executed. By a simple step of demodulation, we now know that the values of $(\texttt{lo}, l_{end})$ are the same in both traces and hence obtain a refutation for the property.

```
 1      func main()
 2      {
 3        const Int h;
 4        const Int h2;
 5        Int[] output;
 6
 7        Int counter = 0;
 8        if(h > 0) {
 9          output[counter] = 5;
10          counter = counter + 1;
11        }
12        else {
13          if (h2 > 0) {
14            output[counter] = 5;
15            counter = counter + 1;
16          } else {
17            output[counter] = 5;
18            counter = counter + 1;
19          }
20        }
21        output[counter] = 7;
22        counter = counter + 1;
23      }
24
```

Figure 4.4: Implicit flow with nested branching.

Example 4. The same reasoning as in example 3 is applied in the proof of the program represented in Figure 4.4. While this program contains nested branches and the proof takes certainly more

reasoning steps than in the example above, we can establish noninterference without any additional trace lemmas. VAMPIREbasically finds a refutation by on the one hand finding that the value of `output[1]` is 7 for any trace and that if the outputs differ in both traces, than `output[1]` cannot be 7 in exactly one of the traces. Thus, the prover is able to derive the empty clause and exclude the possibility of the output differing in the traces.

4.1.3 Noninterference for security

```
1   func main()
2   {
3     const Int blength;
4     const Int n;
5     const Int[] b;
6     const Int a;
7     Int c = 0;
8     Int d = 1;
9     Int i = blength;
10
11    while (i >= 0){
12      i = i - 1;
13      c = 2 * c;
14      d = (d * d) mod n;
15
16      if (b[i] == 1) {
17        c = c + 1;
18        d = (d * a) mod n;
19      }
20      else {
21        skip;
22      }
23    }
24  }
25
```

Figure 4.5: Implicit flow with a loop.

Example 5. The last example in this section combines reasoning we have seen above seemingly with arithmetic reasoning - particularly interesting with regards to security applications. Figure 4.5 represents a program that emulates RSA exponentiation where variables `i`, `blength` $\in L$. Hence, we prove that if the values of `blength` and `i`, are equal in the beginning, then they are also equal after the computation:

$$EqTr(i, l_0) \wedge EqTr(blength) \rightarrow EqTr(i, l_{end}). \tag{4.8}$$

However, none of the variables in L are actually involved in the arithmetic part of the program. Consequently, this example does actually not require any arithmetic reasoning and noninterference

can be established with the help of the same two lemmas as in example 1. Particularly lemmas 4.4 and 4.3 need to be instantiated for `i` to handle induction over the value equality in the traces for the loop at location l_{11}.

Since the equality of `blength` is given by property 4.8 and `i` takes the value of `blength` right at the beginning of the execution, we don't need to reason about any arithmetic expressions in the program, thus allowing VAMPIREto apply the same reasoning as can be seen in example1.

As we have seen, we only need at most two lemmas that are automatically instantiated according to the program structure by RAPID. This allows to reason over a number of noninterference properties for programs containing loops in an automated way.

4.2 Sensitivity

4.2.1 Differential Privacy

Related to the notion of noninterference, we will now discuss differential privacy. Basically differential privacy [Dwo11] defines the property of preserving privacy up to a certain bound, that is allowing a certain loss of privacy without compromising the confidentiality of the individual. Essentially as a policy, differential privacy comes into play when data mining big quantities of potentially sensitive data. As such, we want to ensure that when a large number of sensitive information is gathered and published, we do not disclose information about an individual of a statistical sample. Intuitively, this means that given a bounded variation in input, the published output also varies at most up to this bound. Thus we define differentially private computation as a program given inputs that differ up to a bound k, the "published" output may only differ in k as well.

Formally, we express differential privacy in trace logic $\mathcal{L}$ as follows:

Let l_0 denote the timepoint before the program execution and let $EqTr(v, tp)$ denote that variable v has the same value in both traces at timepoint tp analogously to section 4.1.

Moreover, let $EqTrUpToK(v, tp)$ denote that variable v differs in value in both traces by k at timepoint tp, precisely:

$$EqTrUpToK(v, tp, k) := \begin{cases} \forall pos_{\mathbb{I}}.v(tp, pos, t_1) \simeq v(tp, pos, t_2)) + k & \text{if } v \text{ is a mutable array} \\ \forall pos_{\mathbb{I}}.v(pos, t_1) \simeq v(pos, t_2)) + k & \text{if } v \text{ is a constant array} \\ v(tp, t_1) \simeq v(tp, t_2)) + k & \text{if } v \text{ is a mutable variable} \\ v(t_1) \simeq v(t_2) + k & \text{if } v \text{ is a constant variable} \end{cases}$$

Now, let $L_k \subseteq L$ define the set of the inputs whose values differ by k and $output$ denote the result of the computation differing in both traces at the end by k. Then we define differential privacy as follows:

$$\forall k^{\mathbb{I}} . \Big(\bigwedge_{v \in L \setminus L_k} EqTr(v, l_0) \wedge \bigwedge_{z \in L_k} EqTrUpToK(z, l_0, k) \Big) \rightarrow \Big(\bigwedge_{v \in L \setminus L_k} EqTr(v, l_{end}) \wedge EqTrUpToK(output, l_{end}, k) \Big). \quad (4.9)$$

Note that we use the distinct variable *output* instead of relying on the inputs z being transformed by the computation to emphasize that the output variable might differ from the inputs.

4.2.2 Sensitivity

As a specific mechanism of differential privacy, we particularly investigate sensitivity - a measure of distance within the lines of differential privacy for functions concerned with the question of how far function results differ given similar but not equal inputs [DMNS06, RP10]. Both notions have been discussed extensively in recent literature as in [BGG+16, BGA+14, BEG+17]. Specifically sensitivity denotes how changes in input affect the output of a program. Formally, Barthe et. al. [BGG+16] defines the sensitivity of a function $f : A \rightarrow B$ relative to some metric of A and B, denoted by δ_A, δ_B respectively, as $\delta_B(f(x_1), f(x_2) \leq k * \delta_A(x_1, x_2)$ for every $x_1, x_2 \in I$ where I denotes the set of inputs. We thus say function f is k-sensitive since the changes in output are parameterized by bound k on the inputs. Hence k-sensitivity bounds the distance between the outputs of more or less similar inputs.

We now look at how to formalize and prove properties of this form for some exemplary programs in our input language $\mathcal{W}$. Further, while the definition above is generalized to a set of inputs that differ, notably in the set L_k, the inputs in our examples generally differ in exactly one program variable in one value for the simplicity of representation, thus being a special case of 1-sensitivity. This can, however, be generalized by adjusting the property that is to be proved. While this might make the problem more complex in the number of needed proof steps, trace reasoning remains the same.

Example 6. Consider the example in Figure 4.6: we compute the sum of integers stored in the array `a` and save the value in variable `x`. As a first step, we want to capture the notion that in case that the arrays are equal in both arrays, the sum will not differ in the end. The corresponding property looks as follows:

$$\Big(EqTr(alength, l_0) \wedge \forall j^{\mathbb{I}}.EqTr(a, j, l_0)\Big) \rightarrow EqTr(x, l_{end}) \quad (4.10)$$

Given how closely this property is related to the notion of noninterference given that we only deal with equalities over the two traces, we have to instantiate the same lemmas for the proof. Notably we instantiate the induction lemma 4.4 for program variables `x` and `i` as these variables are assigned and changed in each loop iteration. Further, as was the case for noninterference, the prover needs to establish the same number of iterations for both traces, thus we need to instantiate lemma 4.3 swith the loop counter variable `i`:

```
 1      func main()
 2      {
 3        const Int[] a;
 4        const Int alength;
 5        Int x = 0;
 6        Int i = 0;
 7
 8        while(i < alength){
 9          x = x + a[i];
10          i = i + 1;
11        }
12      }
13
```

Figure 4.6: Sensitivity of sum computations.

$$\begin{aligned}&\Big((\forall it^{\mathbb{N}}.it < lastIt(t_2) \rightarrow i(l_6(it), t_1) \neq 0)\\&\wedge\, i(l_6(lastIt(t_2)), t_1) = 0\Big) \rightarrow lastIt(t_1) = lastIt(t_2)\end{aligned} \tag{4.11}$$

Similarly to noninterference, by establishing that all values of variables `i` and `x` are equal in all iterations of the two traces within the bounds of the loop, and the number of iterations is the same, we can deduct that `x` is the same at the end of the computation.

Example 7. Consider example 4.7 that computes the sum of two arrays `a` and `b` and stores the result in variable `x`. To enforce a difference in input, we establish in the property that `k` takes the value 1 in the first trace t_1 and 0 in the second trace t_2. We then prove that the result saved in `x` after the cmoputation differs by exactly 1. Formally,

$$\begin{aligned}&\Big(EqTr(alength, l_0) \;\wedge\; EqTr(blength, l_0)\\&\wedge\, \forall j^{\mathbb{I}}\,.\,(EqTr(a, j, l_0) \;\wedge\; EqTr(b, j, l_0))\\&\wedge\, k(l_0, t_1) = 1 \;\wedge\; k(l_0, t_1) = 0\Big)\\&\rightarrow EqTrUpToK(x, l_{end}, 1)\end{aligned} \tag{4.12}$$

.

While the reasoning in this example follows a similar structure as the other hyperproperty examples we discussed so far, there is some added complexity that makes it harder to prove. First, we see that while we still use the same lemmas that we used for the other relational examples, we need some additional lemmas. So to reason over the value equality of `x` over both traces in the first loop, we need to instantiate lemma 4.4 for `x` as well as for the loop counter `i`. The same needs to be instantiated for the second loop, but in this case for the variable `y`. As in the last examples for both loops we need to establish that they have equally many iterations in both traces, thus we instantiate lemma 4.3 for `i` at both loop locations, that is l_{12} and l_{21}.

```
 1      func main()
 2      {
 3        const Int[] a;
 4        const Int[] b;
 5        const Int alength;
 6        const Int blength;
 7        const Int k;
 8        Int x = 0;
 9        Int y = 0;
10        Int i = 0;
11
12        while(i < alength)
13        {
14          x = x + a[i];
15          i = i + 1;
16        }
17
18        x = x + k;
19
20        i=0;
21        while(i < blength)
22        {
23          y = y + b[i];
24          i = i + 1;
25        }
26
27        x = x + y;
28      }
29
```

Figure 4.7: Sensitive array.

However, we still miss a way to establish that x and y are not changed in the second and the first loop respectively. Thus we include this statically inferred knowledge in the program semantics, notably as the following lemmas:

$$\forall tr^{\mathbb{T}} \centerdot \forall it^{\mathbb{N}} \centerdot x(l_{21}(zero), tr) = x(l_{21}(it), tr) \tag{4.13}$$

$$\forall tr^{\mathbb{T}} \centerdot \forall it^{\mathbb{N}} \centerdot y(l_{12}(zero), tr) = y(l_{12}(it), tr) \tag{4.14}$$

where l_{12} denotes the location of the first loop and l_{21} denotes the location of the second loop respectively. These lemmas seem obvious to the user as the variables x, y are not changed throughout the respective loops. However, since this requires inductive knowledge about the loop behavior, we so-to-speak need to nudge the prover at this point.

While these lemmas build the basis for the proof, since we do use arithmetic reasoning for the difference in value at the end of the computation stored in `x`, we need to use theory reasoning. Thus for the proof to work we actually need the following two theory axioms:

$$x_0 + x_1 = x_1 + x_0 \tag{4.15}$$

$$x_0 + (x_1 + x_2) = (x_0 + x_1) + x_2 \tag{4.16}$$

These axioms establish commutativity and associativity of addition provided by VAMPIRE's theory reasoning option `-tha on` which includes the axiomatization of linear arithmetic. They are for example needed to establish clauses of the following form $x(end, t_1) \neq sum(1, x(end, t_2))$.

However, this is still not enough for the prover to establish validity of the problem since theory axioms tend to blow up as we will also see in the following example. So in this case, for VAMPIRE to prove the property with in-built theory reasoning we need to make use of the AVATAR-architecture [Vor14] that basically forwards ground problems to a SAT or SMT-solver in the backend while using propositional naming for the non-ground parts to produce models that help the prover select the next sub-goal. Especially with the use of $Z3$ as SMT-solver this enables discharging ground parts with regards to theory reasoning more easily as SMT-solvers still remain stronger in this area of automated proving. The use of AVATAR has been shown to be quite promising when reasoning in combination with theories. However a major drawback of the architecture is that proofs are generally hard to read and understand due to propositional naming of non-ground clauses. While this is of course not the main goal of an automated prover, it is still important for the development of our approach to understand the reasoning to find appropriate lemmas to guide the prover in automatically proving a large number of examples.

Note that the proof also works without AVATAR by manually adding the above two theory axioms to the problem specification that is passed to the prover and switching in-built theory reasoning off which indicates that the problem lies in the fact that small theory axioms are preferred to be instantiated at some point during saturation. Some first experiments with adjusting the weight-to-age-ratio used during saturation to choose the active clause upon which VAMPIRE resolves next, hence focusing more on age, that is on older clauses, have shown the same tendency. This gives a general indication for future work of our approach as we need to make adjustments to theory reasoning in VAMPIRE with the aim of using the prover for real world verification problems.

4.2.3 Limitations of Superposition-based Theory Reasoning

Example 8. The program in the example illustrated in Figure 4.8 computes the sum of integer values in the array `a` and adds an arbitrary value stored in `z` to this sum. The corresponding program property that is proved is again a 2-safety property, i.e. can be expressed over two sets of traces, constituting sensitivity of the computation. Intuitively for two program traces t_1, t_2 where the arrays `a` share the same integer values and the absolute difference between the corresponding values of `z` in both traces differs by at most `k`, we prove that the sum stored in variable `x` after the computation differs by at most `k`.

```
 1      func main() {
 2        const Int[] a;
 3        const Int alength;
 4        const Int z;
 5        const Int k;
 6        Int x = 0;
 7        Int i = 0;
 8
 9        while(i < alength) {
10          x = x + a[i];
11          i = i + 1;
12        }
13
14        x = x + z;
15      }
16
```

Figure 4.8: Sensitivity and limitations of theory reasoning.

Thus, we prove that no information about sensitive/private data stored in the array can be retrieved by running the program multiple times with different values of public inputs simulated by variable z. Formally, we prove the following property:

$$\begin{aligned}&\Big(\forall pos^{\mathbb{I}} \,.\, EqTr(a, pos, l_0) \;\wedge\; EqTr(k, l_0)\\ &\wedge\, EqTr(alength, l_0)\\ &\wedge\, |z(t_1) - z(t_2)| < k(t_1)\Big)\\ &\rightarrow |x(end, t_1) - x(end, t_2)| < k(t_1)\end{aligned} \tag{4.17}$$

where $pos_{\mathbb{I}}$ specifies that pos is integer-valued, $\mathtt{a}(p, t)$ and $\mathtt{k}(t)$ denote the values of a at position p and k at trace t respectively. Note a, k, z are not parameterized with a timepoint as they are constant variables. Similarly, $\mathtt{x}(it, t)$ represents the value of x at timepoint it in trace t and end specifically refers to the last timepoint after the loop execution.

For proving 4.17, we need to consider and instantiate again two trace lemmas expressing relations among values of program variables at trace t_1 and t_2 as follows:

$$\begin{aligned}&\Big(i(l_9(0), t_1) = i(l_9(0), t_2) \wedge\\ &\forall it^{\mathbb{N}} \,.\, i(l_9(it), t_1) = i(l_9(it), t_2) \rightarrow i(l_9(s(it)), t_1) = i(l_9(s(it)), t_2)\Big)\\ &\rightarrow \forall it^{\mathbb{N}} \,.\, i(l_9(it), t_1) = i(l_9(it), t_2)\end{aligned} \tag{4.18}$$

where l_9 denotes the program location of the loop in `line 9` and $l_9(0)$, $l_9(it)$ respectively denote the program locations before the first and the *it*-th loop iteration.

$$\begin{aligned}&\Big((\forall it^{\mathbb{N}} \centerdot it < \mathit{lastIt}(t_2) \rightarrow i(l_9(it), t_1) < \mathit{alength}(t_1)) \\ &\wedge i(l_9(\mathit{lastIt}(t_2)), t_1) \geq \mathit{alength}(t_1))\Big) \\ &\rightarrow \mathit{lastIt}(t_1) = \mathit{lastIt}(t_2)\end{aligned} \tag{4.19}$$

where *lastIt* denotes the last iteration of the loop, i.e. the first iteration where the loop condition fails.

Note that 4.18 is also instantiated for the program variable `x`. Once again, the lemma intuitively states that if two values in two traces t_1, t_2 respectively are equal at the beginning of the loop (indicated by iteration $zero$) at position l_9, and they remain equal for all following iterations (indicated by $s(it)$), then we can conclude that the values of `i` are the same in all iterations for both traces. In this way, the trace lemma ensures the induction step over the equality of a variable during loop execution in two different traces.

The trace lemma of 4.19 states that there are equally many iterations in both traces. This lemma makes sure that the timepoints it are related to the values stored in the iterator variable `i`. With this lemma at hand, we can automatically prove that `x` is equal up to `line 13` in Figure 4.8.

Our recent results in [BEG+19], show that our approach cannot yet verify Figure 4.8 due to the limited theory reasoning support of the prover. Even the AVATAR-architecture does not allow us to automatically find a proof. In particular, reasoning with linear arithmetic axioms makes proof search in VAMPIRE challenging. However, by controlling built-in theory reasoning similarly to the above example and providing additional theory axioms for linear integer arithmetic, our reasoning in trace logic succeeded in proving (4.17). Particularly, we need to add the following axiom:

$$\forall x_1, x_2, x_3, x_4 \centerdot (x_3 + x_1) - (x_3 + x_2) = (x_1 - x_2)$$

This example illustrates that the explicit encoding of timepoints is itself quite powerful as it can be combined with multiple approaches to theory reasoning, hence also leverage powerful SMT-solvers that are very strong in this regard. Thus coupling our encoding with SMT-solvers might be promising for such properties. However, the program semantics is primarily customized for proving within the superposition calculus and enhancing theory reasoning in VAMPIRE is a promising line of future work.

4.3 Hamming Distance

The following section illustrates one of the major advantages of using superposition-based provers with our approach as we will be faced with a k-safety property involving a quantifier alternation. Consider the simple program represented in Figure 4.9. The program computes the sum of integer values stored in the array `a` by iterating over the array and stores the sum in the variable `hw`. With regards to security, we can interpret `a` as a bit-string, thus the program actually stores the so-called Hamming weight of `a` in the variable `hw`.

As we are still in the relational setting, the aim is to prove the following property over two arbitrary computation traces t_1 and t_2 of Figure 4.9: if the elements of the array `a` in t_1 are

component-wise equal to the elements of a in t_2 except for two consecutive positions k and $k+1$, for some k, and the elements of a in t_1 at positions $k, k+1$ are swapped versions of the elements of a in t_2 (that is, the k-th element of a in t_1 is the $(k+1)$-th element of a in t_2 and vice-versa), then the program variable hw is the same at the end of t_1 and t_2. We formalize this property as

$$\forall k^{\mathbb{I}}.\Big((\forall pos^{\mathbb{I}}.((pos \not\simeq k \wedge pos \not\simeq k+1) \rightarrow a(pos,t_1) \simeq a(pos,t_2)) \;\wedge\; a(k,t_1) \simeq a(k+1,t_2) \wedge a(k,t_2) \simeq a(k+1,t_1) \wedge 0 \leq k+1 < alength) \rightarrow hw(end,t_1) \simeq hw(end,t_2)\Big), \tag{4.20}$$

where $k_{\mathbb{I}}$ and $pos_{\mathbb{I}}$ respectively specify that k and pos are of sort integer $\mathbb{I}$.

The property (4.20) is particularly challenging to verify as it involves a quantifier alternation as the array is unbounded and k is arbitrary and requires theory reasoning over linear arithmetic as well. To emphasize the difficulty in automating such a proof, let's look at a high-level proof idea first.

```
1    func main()
2    {
3      const Int[] a;
4      const Int alength;
5
6      Int i = 0;
7      Int hw = 0;
8
9      while (i < alength)
10     {
11       hw = hw + a[i];
12       i = i + 1;
13     }
14   }
15
```

Figure 4.9: Computing hamming weight.

Basically we first have to split the problem into three sub-goals:

(i) iteration 0 to k: from the first position of the array a up to the i-th iteration that has the same value as k we need to prove that the elements in both traces t_1, t_2 are equal, that is hamming weight hw is equal up to the k-th iteration of the loop.

(ii) iteration k to $k+2$: for those loop iterations where i takes the values of $k, k+1, k+2$, we need to prove that while hw might differ in the traces for the $k+1$-th iteration, they are again equal at the $k+2$-th iteration

(iii) iteration $k+2$ to the last iteration end: it remains to ensure that hw has the same value in both traces in all iterations where i takes the values $k+2$ to the arbitrary end, i.e. $(\mathtt{hw}, i, t_1) = (\mathtt{hw}, i, t_2)$ after the swapped array elements as well.

While we can apply the same inductive reasoning for the first and third interval that we already saw in the last sections on relational reasoning, that is we deduce the equality of hw over both traces at the end of the respective intervals from the equality at the beginning of each interval and its step-wise preservation throughout the respective iterations. Particularly for the second interval, this line of reasoning is broken at the end of the k-th or rather the beginning of the $k+1$-th iteration. Thus we need to use arithmetic reasoning that deals with the addition, specifically we use commutativity of addition to conclude that the equality hw in traces t_1 and t_2 is preserved up to the $k+2$-th iteration. By proving all of these subgoals, we may conclude that the equality of hw is preserved until the end of the loop, that is until the end of the program, thus that property 4.20 is valid.

While the above proof might be natural for humans, it is particularly challenging to automate as one needs not only to relate loop iterations and equality over traces but also (i) split the loop at the right intervals that depend on arbitrary values for k, (ii) apply similar reasoning as for noninterference and sensitivity properties to the first and last interval and (iii) in the face of this, also combine superposition-based reasoning with theory-specific reasoning, particularly to prove the equality after the second interval.

It is thus not surprising, that given the restrictions on reasoning modulo theories with full first-order solver, the proof requires to be split - proving each of the above intervals. Precisely, we first prove the equality up to iteration $k + 2$ and use this proof as a lemma to prove the complete property. Note that while this does indeed have the flavor of interactive theorem proving, this is mostly necessary due to the blow up in theory axiomatization of integers. Moreover, we are not required to prove every step manually but need to guide the automated prover in the right direction by splitting up complex reasoning.

So in order to prove the first part, that is the equality of hw over iterations 0 to $k + 2$, we need to instantiate the inductive scheme that was discussed in previous sections, i.e. we instantiate lemma 4.4 for program variable i. Since Lemma 4.4 is insufficient to reason about trace equality of hw as its premise is violated at iterations k and $k + 1$, we need to adjust the lemma to allow the deduction the equality for hw up to position $k + 2$.

Formally this is achieved with the following lemma:

$$\forall itB^{\mathbb{N}}.\Big((EqTr(hw, \mathtt{0}) \wedge \forall it^{\mathbb{N}}.(it < itB \wedge EqTr(hw, it)) \rightarrow EqTr(hw, \mathtt{succ}(it))) \rightarrow EqTr(hw, itB)\Big). \tag{4.21}$$

Essentially, lemma 4.21 states that for any iteration itB, if the hw is equal at the beginning and for all iterations smaller than itB, we can infer that the equality of hw is preserved from one iteration to the next, we deduce its trace equality in iteration itB as well. This is essential to prove the equality up to iteration k.

The reasoning about the shortly differing values of hw after the k-th iteration, that is the second subgoal, needs theory-specific reasoning. Since we need to reason over sums of two different iterations in the respective traces, that is we need to conclude that hamming weight in iteration k in the first trace t_1 is actually the same as the hamming weight in iteration $k + 1$ in trace t_2, we need to add the appropriate theory axiom manually to allow for Vampire's reasoning under `-tha some`, as full theory axiomatization for integers blows up during saturation. In this case, what we need is associativity of addition:

$$(x_0 + x_1) + x_2 = (x_0 + x_2) + x_1 \tag{4.22}$$

Secondly, what is actually crucial for the proof to work is to specify what iterator positions, that is what natural numbers, respond to the iterations k and $k+2$ that are of sort integer. This can be understood as relating values of sort integer with values of the natural numbers. To do so, we indeed need to specify that the two specific iteration itK and $itkPlus2$ of sort $\mathbb{N}$ correspond to those iterations where i in location l_{12} is equal to the value k and to the value $k + 2$ respectively.

While this specification is added manually to the property we want to prove, we have to define a lemma that lets the prover use this information for the proof up to iteration $k + 2$.

Then the following trace lemma that expresses how iterations are *synchronized* with the integer values of the loop counter variable is enough to finish the proof up to iteration $k + 2$:

$$\forall tr^{\mathbb{T}} \centerdot \left(\forall it_1^{\mathbb{N}}, it_2^{\mathbb{N}} \centerdot it_1 < it_2 \rightarrow i(l_{12}, it_1, tr) < i(l_{12}, it_2, tr)\right). \tag{4.23}$$

In the above way, we can prove the first two subgoals and then use this proof as a lemma for the main goal which follows essentially the same line of reasoning as the first subgoal.

CHAPTER 5

State of the Art

In this chapter we overview the most related approaches to our work in automating program analysis and verification. This chapter is separated in two parts: the first one focuses on different methods for formal verification apart from theorem proving in general. The second part emphasizes particularly the state-of-the-art methods applied to hyperproperty verification.

5.1 Software Verification

Deductive Verification. Based on Hoare Logic [Hoa69], the task of software verification can be deduced to proving the correctness of Hoare triples of the form $\{A\}P\{B\}$ where A and B are formulas that express pre- and postconditions respectively and P is a program. Depending on how the triple is verified, we can express partial correctness or total correctness. Partial correctness of a triple states that for a state σ where A holds, if the execution of P results in a state σ' (i.e. P does not necessarily terminate), σ' satisfies B. Formally,

$$\textit{forall states } \sigma\textit{, s.t. } \sigma \models A \textit{if } \langle \sigma, P \rangle \rightarrow \sigma'\textit{, then } \sigma' \models B \;\; s \tag{5.1}$$

Similarly we can express this claim for total correctness which extends partial correctness with termination resulting in: for any state σ where A holds, the execution of P results in σ', st. $\sigma' \models B$ holds. Formally, we have

$$\textit{forall states } \sigma\textit{, if } \sigma \models A\textit{, then } \langle \sigma, P \rangle \rightarrow \sigma' \land \sigma' \models B \tag{5.2}$$

Hence, proving correctness of programs results in sequentially defining such triples for all execution steps of a given program. Based on this paradigm, we use syntactic transformations, so-called *predicate transformers* based on the work of Dijkstra [Dij78] to generate the necessary verification conditions, i.e. first-order formulas, according to the structure of the program that then need to be discharged either manually or as is more common by a potentially automated theorem prover. Proving the validity of all proof obligations constitutes the validity of the given Hoare triple with regards to the (partial or total) correctness claim.

Satisfiability Modulo Theories. Many automated tools in the area of deductive verification are based on SMT-solving. Satisfiability modulo theories is a decision problem for (quantifier-free) first-order formulas with respect to reasoning with different background theories. Common background theories are the theory of linear arithmetic (integers and reals), the theory of arrays, bit vectors or other common (algebraic) data structures, as well as the theory of equality and uninterpreted functions (EUF). Z3 [DMB08] and CVC4 [BCD$^+$11] are state-of-the-art SMT solvers that incorporate a number of decision procedures to enhance results.

SMT-Solving is based on common decision procedures for SAT-solving. Early approaches to SMT-solving [ACG99, ABC$^+$02, BDS02, DMRS02] were based on identifying a propositional model with a SAT-solver and then use theory-specific solvers to prove the consistency of the given model within the background theory. Now, there are multiple approaches following these lines and often differentiate in when to call the theory solver on a given model, i.e. after a full model has been found or throughout the SAT-solvers procedure to reduce the depth of necessary backtracking in case a conflict is found.

Most modern SMT-solvers are based on the DPLL($\mathcal{T}$) decision procedure [Tin02] which combines theory reasoning with the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60, DLL62] for SAT-solving. The SAT-based DPLL-algorithm is a depth-first search to prove satisfiability of a ground (quantifier-free) logical formula in conjunctive normal form (CNF) by trying to find a model by consecutively assigning a truth value to a chosen literal. Unit propagation, that propagating the truth value of so-called unit clauses, which are clauses that contain only a single literal, is used to prune the search space. Precisely, for a given set of clauses, we apply unit propagation with backward subsumption, hence removing redundant clauses subsumed by smaller (unit) clauses until all variables are assigned. Thus a model that proves satisfiability could be constructed, or the empty clause $\square$ is derived, hence showing that the original clause set is unsatisfiable. The idea is to keep track of a partial assignment and extend this assignment until all variables are assigned or a conflict, that is a logical contradiction, is reached. In case of a conflict the algorithm backtracks and branches differently, i.e. changes the truth value of a certain assignment according to the chosen branching heuristics. A lemma excluding the possible conflict is then added to make sure that it will not happen again. This last approach is commonly referred to as conflict-driven clause learning in the literature. This procedure is complete, thus, will either prove or disprove satisfiability of a propositional/ground clause set.

SMT-solvers extend this idea for theory reasoning. Based on the theoretical results captured in a sequent-style calculus by Tinelli [Tin02], a general architecture for the combination of theory solvers and SAT-solvers is given by the DPLL(T) [GHN$^+$04] framework. Note that sometimes the core SMT-system is referred to as CDCL(T) instead of DPLL(T) highlighting that conflict-driven clause learning is widely applied among standard SAT and SMT-solvers. The main difference to early approaches is that theory reasoning is built-in in the DPLL decision procedure. In other words, the background theories are used to decide satisfiability instead of finding a propositional model and checking its consistency with the theory in hindsight.

On the theoretic level, to give an idea how this works, we will examine how unit resolution with backward subsumption is used in this way. The idea is expressed in a sequent calculus-style as in [Tin02] as follows: for a given set of clauses $\{\phi, l \wedge C\}$, a context Λ used to store assigned

literals and a background theory $\mathcal{T}$, such that $\Lambda \vdash \phi, l \wedge C$, if we want to resolve on literal l, the necessary side condition is that $\neg l$ is entailed by the background theory in a given context, i.e. $\Lambda \models_{\mathcal{T}} \neg l$ holds, yielding $\Lambda \vdash \phi, C$ after resolving on literal l. Note that the difference to SAT-procedure lies in the side condition asserting that $\neg l$ is entailed in the theory instead of the side condition being that $\neg l$ is in the set Λ of already asserted literals. This way any asserted literal is also already asserted in the theory instead of having to check with a theory solver at a later point. This theoretical framework was firstly applied in the aforementioned DPLL(T) framework by [GHN$^+$04] for the theory of equalities and uninterpreted functions (EUF) which relies on interfacing a theory solver for T in a way to drive the search for a satisfying model. While the work is practically based on integrating a solver for EUF, the framework has been interfaced with multiple theories, most notably with a solver for the theory of linear arithmetic in [DDM06]. Since then theory reasoning has been made possible for multiple theories, namely EUF, linear arithmetic, fixed-size bit-vectors, the theory of arrays.

Note that these approaches mainly target theories with decidable decision procedures, i.e. ground (quantifier-free) fragment of first-order logic. While there are some attempts to reason in undecidable fragments of logic such as non-linear arithmetic e.g. in [JdM12], most SMT-solvers work in decidable fragments of many-sorted first-order logic with built-in theories. Hence, they are often limited in their expressiveness as quantifier alternation is in general not expressible which is one of the major drawbacks compared to the use of full first-order logic and superposition-based reasoning.

However, this is a major limitation for software verification as many specifications cannot be expressed without universal quantification. Particularly for reasoning over loops in a static way it is often necessary to use universal quantification to express invariants needed to discharge conditions. Further, many theory axiomatizations rely on universal quantification, such as the theory of linear arithmetic. To this end, modern SMT-solvers allow for quantifier instantiation, i.e. instantiating a universal formula to ground instances in the hopes of finding the necessary one to prove unsatisfiability/validity during proof search. This allows at least for the use of "simple" formulas involving quantifiers, i.e. without quantifier alternation(s). In state-of-the-art SMT-solvers like Z3 and CVC4 quantifiers are handled by so-called E-matching abstract machines [DMB07] based on a matching algorithm that matches existing ground terms in the conjecture with sub-terms of the quantified formula to find appropriate instances for further resolving. Note that this is possible since any formula $\forall \overline{x}.F$ can be interpreted as an infinite conjunction over all possible substitutions θ for $\overline{x}$: $\bigwedge_\theta \theta(F)$.

While this allows for higher degrees of automation, the user of such systems is still required to come up with their own loop invariants to prove different specifications about programs involving loops and is restricted to formulas without quantifier alternations.

SMT-Solving and Intermediate Verification Languages. A noteworthy application of SMT-solving along the lines of this research that needs to be addressed is software verification with an intermediate verification language and an SMT-solver to discharge verification conditions. One prominent example in this line of work is the language and verifier *Dafny* [Lei10] using a combination of the intermediate verification language *Boogie* and SMT-solver *Z3* for automatically proving functional correctness.

Dafny is a typed imperative programming language that allows for automated deductive verification for total correctness by providing the user with built-in specification mechanism for pre- and postconditions, loop invariants, variants for termination in the style of Hoare logic and some other helpful features such as ghost variables that help in the specification process. *Dafny* code is then compiled to the *Boogie 2* [Lei08, BCD+05] intermediate verification language (IVL), that is the semantics of Dafny programs are based on the semantics of the Boogie IVL.

Boogie further provides a tool to generate proof obligations in first-order logic that are passed to SMT-solver Z3 to be discharged. Note that the translation from a Dafny to a Boogie program is sound in the sense that a correctness proof of the Boogie code implies the correctness of the original Dafny program. Dafny has been proven to be very powerful and is able to prove pointer-based programs as it allows for definitions of (algebraic) data types and function calls. Apart from that, Boogie 2 comes with a type system [LR10] allowing for static type checks of built-in and user-defined types.

All in all the integrated Dafny environment can already be seen as a very powerful verification suite. However, while it offers many features, a major difference compared to RAPID is found in the program semantics: Dafny/Boogie's program semantics program semantics are based on standard Hoare logic/Dijkstra's predicate transformers while RAPID's semantics extend this idea with explicit timepoints - standard Hoare Logic can be seen as an instantiation of Trace Logic. The power of this new approach to semantics allowed us to prove a rather simple set of programs – the break examples at section 3.2 – that traverses an array and breaks the while-loop once a specific element has been found while the Boogie 2 specification needed an invariant specified by the user to prove functional correctness of the program.

The combination of semantics with powerful trace lemmas therefore is able to prove a simple program out of the box where other approaches still need appropriate invariants that handle the inductive part of reasoning over loops. While Dafny/Boogie leverages the power of SMT-solvers, one can understand the combination of RAPID with first-order solver Vampire as a new approach for automated software verification enhancing the degree of automation and expressiveness in the sense that our approach (1) leverages powerful first-order solvers with theories and (2) provides more fine-grained program semantics by making program timepoints explicit paving the way for automating inductive reasoning. This entails multiple new problems such as the incompleteness of theory reasoning in the non-ground case that SMT-solvers don't face but also offers new opportunities such as (1) automatically proving complex properties with quantifiers as well as (2) decreasing the need of an expert user to find appropriate invariants/variants to make software verification an integral part of the development process of highly-secure and "at risk" systems.

Another prominent example is the combination of the `WhyML` verification language and the `Why3` tool [FP13] for deductive verification. `Why3` makes use of multiple automated (Simplify, CVC as well as interactive theorem provers (Coq, Isabelle/HOL, HOL etc.) to discharge verification conditions. `WhyML` is a simple imperative programming language equipped with a type system based on `ML` - comparable to *Boogie 2*. However, for verification reasons language features are limited to first-order reasoning, thus, do not include higher-order functions. As with *Boogie 2*, `Why3` requires the user to specify pre- and postconditions as well as invariants for reasoning over loops. By providing variants also termination can be proved. Proof obligations are generated by standard weakest-precondition transformers. `Why3` also allows for extraction

of executable code by providing a translation from WhyML specifications to OCaml executables. Thus, both approaches have similar advantages and drawbacks in comparison with our approach.

The KeY-Project [BHS07, ABB+16] is another example of functional verification but also extends to analysis of information flow. Since the KeY framework originated as a tool for Java, it is now mainly used for Java source code in combination with annotations in the Java Modelling Language (JML) [LBR98]. The intermediate verification language of the tool is Java Dynamic Logic [Bec00] - hence the proof calculus which is based on sequent calculus adheres to Java semantics. The KeY prover environment tries to prove the verification conditions automatically with its own prover as well as external sources such as SMT-solvers. In case that no proof was found, the user can also guide the proof search by manuelly guiding the proof steps such as in interactive theorem proving. The project is thus more guided by industry-needs instead of theoretical research.

Model Checking for Software Verification. Another standard technique for formal verification is model checking [CJGK+18]. The idea relies on interpreting programs as finite state transition systems and their specifications as temporal properties formalized with temporal logics such as CTL*, CTL or LTL. In case of showing functional correctness, the model checker ensures that there is no path in the transition system such that the property gets falsified. In case such a path exists, the model checker reports a counterexample to the given property.

Model checking applies to both safety and liveness properties. The former ensures that something bad will never happen, i.e. there's no execution path that is a counterexample to the safety property - an unsafe state can never be reached. The latter is employed to ensure that something good is going to happen at some point in the future, i.e. we will always reach a state such that the property holds.

A major problem for liveness properties lies in the infinity of the property that something will *always* happen since efficient model-checking methods mainly apply to finite state systems. While there exist approximative methods to turn an infinite system into a finite one, the problem of finding a counterexample for debugging is far easier than proving that no counterexample exists. The challenge here originates from the size of the given state transition system. While there are methods to abstract an infinite transition system into a finite one (with some loss of precision) e.g. in [CGL94], adding state variables to standard state transition systems still leads to a major blowup. Hence, model checking is faced with what is commonly referred to as the *state explosion problem*. Thus, modern model checkers are equipped with a multitude of mechanisms for abstraction of states such as interpolation [McM03] and counterexample guided abstraction refinement (CEGAR) [CGJ+00] and ultimately bounded model-checking (BMC) [BCC+03].

The latter leverages the power of SAT-solvers by encoding systems and properties in propositional logic to find counterexamples of bounded length, that is in some k transitions of the model. Particularly, for loops this is an interesting approach since loops are iteratively unrolled until a counterexample is found. Hence the BMC allows for efficient bug finding in programs with loops. However the limitation of bounded loops makes this method incomplete for proving functional correctness. This method is also employed in state-of-the-art model checkers such as CBMC [KT14] and Alloy [VD12]. There are also some approaches such as [MK11] using SMT-solvers which allows to express properties in combination with theories.

While these abstractions make model checkers very efficient in practice, they introduce the problem of imprecision due to overapproximation - specifically overapproximation of the post-image (overapporoximated set of reachable states from some state). Hence, abstraction might lead to spurious counterexamples which introduces the need to backtrack and refine the level of abstraction in order to find a true counterexample (or exclude the existence of it).

One of the major advantages of all these techniques is that they are fully automated since they work on a finite domain and are in case of failure ready to provide an execution path that leads to a counterexample - in general a hard task for theorem proving. Hence the main application of model-checking approaches lies in debugging software (and hardware) systems. However, due to the limitation to finite systems as well as the state explosion problem, model-checking is usually not suited for purely proving functional correctness and thus exists alongside other formal techniques.

Abstract Interpretation and Static Analysis. Abstract interpretation was firstly introduced in the 70ies by Patrick and Radhia Cousot [CC77, CC92] and aims at proving properties over runtime program behavior in a static manner, that is without executing the code. The main applications lie in type, control flow and data flow analysis to show for example that programs are free of runtime exceptions, such as nullpointers and buffer overflows or that a program terminates. Many such methods are applied to compiler optimization and bug finding tools.

Abstract interpretation provides a theory for correctly approximating program behavior in order to automate an analysis task. An example for this is type checking where a set of type equality constraints are established from a given syntax tree of a program such that all expressions and variables are evaluated under the semantics. The program is considered to be typable if it satisfies those constraints (which can in this case be checked by unification).

Formally, for a given programming language L and an abstract domain $\mathcal{D}$, we need to define semantics S of the language such that $S[\![p]\!] \in D$, that is each expression $p \in L$ obtains a semantic value. Further, one has to define an abstract domain $\mathcal{D}^\#$ and an abstract semantics $S^\#$ such that $S^\# \in L \to \mathcal{D}^\#$. To prove soundness for an abstract interpretation of program P we need to show that abstract semantics $S^\#$ is computable for each expression $p \in P$, i.e. that $S[\![p]\!] \in \{S | \sigma(S, S^\#[\![p]\!])\}$ where σ is a soundness relation that necessarily satisfies for all expressions $p \in L$ that $\sigma(S[\![p]\!], S^\#[\![p]\!])$. Since all of the mentioned program properties are known to be undecidable by Rice's famous theorem, abstract interpretation is a very hard task that requires finding sound approximations for program semantics and abstract domains that are coarse enough to prove many programs correct but also conservative enough to remain sound. Hence, one of the challenges in approximating program behavior is that it might lead to false negatives, that is the rejection of correct programs based on the violation of some constraints that in the end were too conservative to prove the program correct. Hence balancing overapproximation for problem-specific domains is an widespread and still ongoing research area that is based on finding useful abstract domains for many different problems. This task can be related to our efforts in finding problem-specific sets of trace lemmas that can automatically be instantiated for programs with properties of a certain kind. Hence one can understand the search for such lemmas as finding the abstract domain for superposition-based automated theorem proving.

5.2 Relational Verification

Hyperproperty Verification. Given the rise of using formal methods for security properties, relational verification is becoming a hot topic in software verification research. Hyperproperties are generally properties that are expressed as sets of sets of execution traces, i.e. they relate multiple execution traces with each other. Since many security properties are in general such relational properties, we will look into the current methods for relational verification of noninterference, sensitivity and other k-safety properties as there exist multiple approaches for various properties based on the methods defined above.

Type Systems. Starting with static analysis, Sabelfeld [SM03] propose type systems for analyzing noninterference in programs and protocols that annotate program variables and expressions with security types, e.g. *high* and *low* to refer to the level of confidentiality. On a policy level, the type system ensures that low-level inputs don't interfere with highly confidential outputs, that is an attacker having knowledge of any *low* inputs/outputs cannot make assumptions about any *high* data. One of the major drawbacks of this method is its limitation to static analysis that dismisses many safe programs as potentially unsafe due to overapproximation. For instance, the above mentioned method fails to prove our example3 since *high* program variables are generally disallowed for soundness reasons.

Type systems are successfully applied to various properties concerning information security like secrecy and authentication in cryptographic protocols such as [BFG+14, CEK+15, CGLM17, CGLM18] e.g. for the verification of electronic voting systems. Applications of these methods mostly concern equivalence/indistinguishability properties, for instance anonymity of a protocol where we want to verify that an attacker cannot differentiate between a protocol session of Alice or Bob. Type systems for these kinds of properties have been proven more efficient than standard tools of protocol verification such as ProVerif [B+01, BAF08] based on Horn clause resolution or term-rewriting based tool Tamarin [MSCB13]. While we're dipping our feet into verification of hyperproperties, RAPIDis not yet fully set out for protocol verification. Adapting the input modelling language with some extensions for function calls and a session model might be an interesting line of future work that would allow handling protocol verification and might speed up current results on an unbounded number of executed sessions.

Relational Hoare Logic. Another interesting aspect that relates to our ongoing research efforts is relational Hoare logic [Ben04], self-composition [BDR04] and the use of product programs [BCK11] for verification of hyperproperties. Relational Hoare logic gives a theoretical extension of Hoare triples to Hoare quadruples for deductive verification of two programs or two executions of the same program with the same termination behavior but is limited in expressiveness as properties involving quantifier alternations are generally not captured.

Based on composing two different programs or two runs P_1, P_2 of a program in a sequential manner into a single program P, self-composition reduces the task of proving a property for the compositional program in a standard non-relational way based on Hoare triples. Product programs extend this idea by simulating the execution of two programs simultaneously in lock step. This approach was extended in [BCK13] for asymmetric products, that is for two different programs P_1, P_2, we want to check that for all traces of P_1, there exists a trace in P_2 where some

property holds. However, while especially the last approach seems very promising to cover many security properties as non-determinism can be captured as well, all of the approaches suffer from lack of expressivity when it comes to k-safety properties for more than two program traces. Also all of these approaches are based merely on the syntactical content of a program.

HyperLTL. Checking hyperproperties with temporal logics and model-checking algorithms is also ongoing research: HyperLTL [CFK+14] is an extension of linear temporal logic with so-called trace quantifiers allowing to relate multiple execution traces of a program. Thus, HyperLTL enables proving information flow properties such as noninterference which is captured by the following formula F:

$$\forall \pi, \pi' \,.\, \Big(G \bigwedge_{i \in I \setminus \{h\}} i_\pi \leftrightarrow i_{\pi'} \Big) \rightarrow G(o_\pi \leftrightarrow o_{\pi'})$$

where π, π' are the respective execution traces, the set I is the set of inputs, o are the respective outputs and h is an input value of high security value, that is a secret variable, and $G\phi$ is a temporal operator intuitively specifying that the formula ϕ holds everywhere, hence *globally*, on the subsequent path. Hence F specifies that for all execution traces, if the *low* security input values are equal everywhere but in the *high* security context, then also the outputs will be the same. Tools such as in [FRS15] have been developed to use model-checking algorithms on specifications such as above. While HyperLTL was not yet fit to prove properties with quantifier alternation, the recent work [CFST19] has been shown to be very expressive and allows handling of HyperLTL-formulas with one quantifier alternation. They particularly established a proof-of-concept by proving generalized noninterference, formally expressed as follows:

$$\forall \pi, \pi' \,.\, \exists \pi'' \,.\, \Big(G(h_\pi \leftrightarrow h_{\pi''}) \wedge G(o_{\pi'} \leftrightarrow o_{\pi''}) \Big)$$

The property specifies that the public outputs of traces π and π' don't depend on any value of h simulated by existential quantification on a third trace π'' used to "inject" high-security values. The work relies on a game theoretic approach that creates two players for universal and existential quantification respectively such that the existential player has to match every move made by the forall-player which implies satisfiability of the formula at hand. Formally, they can prove formulas of the form $\forall \pi \,.\, \exists \pi' \,.\, \phi$, where a winning strategy of the existential player implies that ϕ is satisfied by the pair (π, π'). Every move made can be interpreted as stepwise computation of an execution trace. This is further automatically enabled by eliminating this existential quantification with the help of known synthesis algorithms. While this is an impressive result, HyperLTL remains highly undecidable - there are no decision procedures for the satisfiability problem of the $\forall\exists$ fragment [FH16] - and the model-checking problem on finite Kripke structures and HyperLTL formulas is non-elementary in terms of worst-case complexity.

F/Interactive theorem proving.* Another popular approach is using interactive theorem provers such as Coq, Isabelle/HOL. As for security properties, F* [SHK+16] is a state-of-the-art prover of relational properties by the use of refinement types. While this approach is promising for relational verification, it remains for the user to find program-dependent invariants/lemmas to find a proof which is not required with our approach.

CHAPTER 6

Conclusion

6.1 Conclusion

Based on the new framework RAPID that encodes imperative programs in trace logic $\mathcal{L}$, an instance of full first order logic, we show how to automatically discharge verification conditions with superposition-based first-order automated theorem provers, in particular the VAMPIRE theorem prover. To this end, we illustrated how inductive reasoning over loops with arrays is automated by instantiating a set of trace lemmas, allowing us to express inductive loop properties, that are more general than loop-specific invariants. By doing so, we do not rely on users to provide loop specific invariants.

The major driver behind this approach is the custom encoding that allows to explicitly express timepoints as functions over program locations and loop iterations. This allows to express properties in such a way that we can address loop iterations and quantify over them as pleased, that is we can add to our encoding assertions over *all* or *some* loop iterations. Furthermore, we can reason about arbitrary program values of integer and array variables. The combination of our encoding in $\mathcal{L}$ with full first-order theorem provers equips our work with four major advantages:

(i) *Loop nesting.* A major advantage of the formalism is its simple extension for nested loops. By allowing to extend timepoints over multiple iterations according to the program structure, all trace lemmas can be generated for arbitrarily nested loops, thus allowing to reason over such loops in the same way.

(ii) *Relational verification.* By extending the encoding of timepoints with traces, we can also handle so-called hyperproperties relating multiple sets of traces. With this approach we showed how to prove common security properties such as noninterference and sensitivity.

(iii) *Quantifier Alternation.* VAMPIRE's superposition-based first-order reasoning allows us to prove complex properties that possibly involve quantifier alternations as can be seen in 3.3 for single trace reasoning and in 4.3 for relational verification.

(iv) *Automated Trace Reasoning.* The automatic instantiation of trace lemmas over all relevant program variables allows us to prove properties that would otherwise need inductive reasoning manually handled by proving with interactive theorem provers or equipping an automated prover with a program-specific invariant. While statically inferred from semantics, thus requiring some manual work, once found, they allow us to prove sets of problems instead of a specific property about a single program.

Our results show that the combination of RAPID and VAMPIRE provide a promising tool chain for proving in particular hyperproperties 4. Results of our recent paper [BEG+19] showed that RAPID's semantics is more expressive than state-of-the-art noninterference verification tools and that Vampire is better suited to the verification of security-relevant hyperproperties such as noninterference and sensitivity than state-of-the-art SMT-solvers like Z3 and CVC4.

6.2 Challenges and Future Work

Ongoing work focuses on designing (theory-)specific inference rules in the superposition calculus that identify redundant inequalities during proof search. We also intend to improve clause selection during proof search. As trace lemmas might in general be quite long formulas, their assigned weights are usually much higher than those of small theory axioms such as axioms of linear arithmetic. Thus, at a certain point during proof search, trace lemmas are not used anymore as smaller clauses (such as theory axioms) are usually preferred by superposition-provers, and hence VAMPIRE. Adjusting clause selection to use trace lemmas at arbitrary steps during proof search is a challenging task we aim to address.

Another idea is to extend the superposition calculus to reasoning over inequalities, as this would allow to leverage the power of superposition-based inferences for inequality reasoning. Note that many arithmetic inequalities of the form $s(0) > 0$, $s(s(0)) > 0$, ... are generated during saturation. While they mostly get handled by redundancy, hence do not enter the *active clause set* - that is the set to be resolved upon - they still pose a threat to efficiency of the prover and thus also finding a proof in a given time limit. Superposition over inequalities might help to make logically "stronger" inferences. Future experiments need to be made to evaluate the impact of such rules.

Another line of work is the extension of our simple input language and thus program semantics to more complicated constructs like function calls. At the moment, we handle exactly one function with arbitrary loop nesting. Extending the semantics would allow us to look into complex examples with wide applications to security verification such as verifying smart contracts.

List of Figures

Bibliography

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. Deductive software verification–the key book. *Lecture Notes in Computer Science*, 10001, 2016.

[ABC+02] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A sat based approach for solving formulas over boolean and linear mathematical propositions. In *International Conference on Automated Deduction*, pages 195–210. Springer, 2002.

[ACG99] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. Sat-based procedures for temporal reasoning. In *European Conference on Planning*, pages 97–108. Springer, 1999.

[B+01] Bruno Blanchet et al. An efficient cryptographic protocol verifier based on prolog rules. In *csfw*, volume 1, pages 82–96, 2001.

[BAF08] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *The Journal of Logic and Algebraic Programming*, 75(1):3–51, 2008.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[BCD+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael J. Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick,*

Ireland, June 20-24, 2011. Proceedings, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011.

[BCK13] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*, volume 7734 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2013.

[BDR04] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004.

[BDS02] Clark W Barrett, David L Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to sat. In *International Conference on Computer Aided Verification*, pages 236–249. Springer, 2002.

[Bec00] Bernhard Beckert. A dynamic logic for java card. In *Workshop on Formal Techniques for Java Programs (FTfJP). Technical Report*, volume 269. Citeseer, 2000.

[BEG+17] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving expected sensitivity of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 2(POPL):57, 2017.

[BEG+19] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovacs, and Matteo Maffei. Verifying Relational Properties using Trace Logic. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 170–178, 2019.

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 14–25. ACM, 2004.

[BFG+14] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 193–205. ACM, 2014.

[BGA+14] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. Proving differential privacy in hoare logic. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 411–424. IEEE, 2014.

[BGG+16] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving differential privacy via probabilistic couplings. In *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10. IEEE, 2016.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

[CEK+15] Véronique Cortier, Fabienne Eigner, Steve Kremer, Matteo Maffei, and Cyrille Wiedling. Type-based verification of electronic voting protocols. In *4th International Conference on Principles of Security and Trust - Volume 9036*, pages 303–323. Springer-Verlag, 2015.

[CFK+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014.

[CFST19] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In *International Conference on Computer Aided Verification*, pages 121–139. Springer, 2019.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[CGL94] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[CGLM17] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. A type system for privacy properties. In *2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 409–423. ACM, 2017.

[CGLM18] Veronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. Equivalence properties by typing in cryptographic branching protocols. In *Principles of Security and Trust (POST'18)*, pages 160–187. Springer, 2018.

[CJGK+18] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. 2018.

[DDM06] Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for dpll (t). In *International Conference on Computer Aided Verification*, pages 81–94. Springer, 2006.

[Dij78] Edsger W Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. In *Programming Methodology*, pages 166–175. Springer, 1978.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DMB07] Leonardo De Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *International Conference on Automated Deduction*, pages 183–198. Springer, 2007.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[DMRS02] Leonardo De Moura, Harald Rueß, and Maria Sorea. Lemmas on demand for satisfiability solvers. *Proc. SAT*, 2:244–251, 2002.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[Dwo11] Cynthia Dwork. Differential privacy. *Encyclopedia of Cryptography and Security*, pages 338–340, 2011.

[FH16] Bernd Finkbeiner and Christopher Hahn. Deciding Hyperproperties. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory (CONCUR 2016)*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3?where programs meet provers. In *European Symposium on Programming*, pages 125–128. Springer, 2013.

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking hyperltl and hyperctl ^*. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 30–48. Springer, 2015.

[GHM13] Jürgen Graf, Martin Hecker, and Martin Mohr. Using joana for information flow control in java programs-a practical guide. *Software Engineering 2013-Workshopband*, 2013.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll (t): Fast decision procedures. In *International Conference on Computer Aided Verification*, pages 175–188. Springer, 2004.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[JdM12] Dejan Jovanović and Leonardo de Moura. Solving non-linear arithmetic. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 339–354, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[KRV17] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to terms with quantified reasoning. In *POPL*, pages 260–270. ACM, 2017.

[KT14] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[KV13] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, pages 1–35, 2013.

[LBR98] Gary T Leavens, Albert L Baker, and Clyde Ruby. Jml: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA?98)*, pages 404–420. Citeseer, 1998.

[Lei08] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[Lei10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[Lei12] K Rustan M Leino. Automating induction with an smt solver. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 315–331. Springer, 2012.

[LR10] K Rustan M Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 312–327. Springer, 2010.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[McM03] Kenneth L McMillan. Interpolation and sat-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003.

[MK11] Aleksandar Milicevic and Hillel Kugler. Model checking using smt and theory of lists. In *Nasa Formal Methods Symposium*, pages 282–297. Springer, 2011.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*, pages 696–701. Springer, 2013.

[RP10] Jason Reed and Benjamin C Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *ACM Sigplan Notices*, volume 45, pages 157–168. ACM, 2010.

[SHK+16] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In *POPL*, pages 256–270, 2016.

[SM03] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE J. on Selected Areas in Communications*, 21(1):5–19, 2003.

[Tin02] Cesare Tinelli. A dpll-based calculus for ground satisfiability modulo theories. In *European Workshop on Logics in Artificial Intelligence*, pages 308–319. Springer, 2002.

[VD12] Amirhossein Vakili and Nancy A Day. Temporal logic model checking in alloy. In *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pages 150–163. Springer, 2012.

[Vor14] Andrei Voronkov. Avatar: The architecture for first-order theorem provers. In *Proceedings of the 16th International Conference on Computer Aided Verification-Volume 8559*, pages 696–710. Springer-Verlag, 2014.