Verifying Relational Properties using Trace Logic

Gilles Barthe[*,†], Renate Eilers[‡], Pamina Georgiou[‡], Bernhard Gleiss[‡], Laura Kovács[‡,§], Matteo Maffei[‡]
[*]Max Planck Institute for Security and Privacy, Germany
[†]IMDEA Software Institute, Spain
[‡]TU Wien, Austria
[§]Chalmers University of Technology, Sweden

***Abstract*—We present a logical framework for the verification of relational properties in imperative programs. Our framework reduces verification of relational properties of imperative programs to a validity problem in trace logic, an expressive instance of first-order predicate logic. Trace logic draws its expressiveness from its syntax, which allows expressing properties over computation traces. Its axiomatization supports fine-grained reasoning about intermediate steps in program execution, notably loop iterations. We present an algorithm to encode the semantics of programs as well as their relational properties in trace logic, and then show how first-order theorem proving can be used to reason about the resulting trace logic formulas. Our work is implemented in the tool RAPID and evaluated with examples coming from the security field.**

I. INTRODUCTION

Program verification generally focuses on proving that all executions of a program lie within a specified set of executions, that is, properties are seen as sets of traces. However, this approach is not general enough to capture various fundamental properties, such as non-interference [1] and robustness [2]. These notions are naturally modelled as relational properties, that is as properties over sets of pairs of traces. Relational properties are special instances of hyperproperties [3], which are formally defined as sets of sets of traces.

Verification of relational properties can be achieved in different ways. One approach is by reduction to program verification: given a program P and a hyperproperty ϕ, construct a program Q and a property ψ, such that: (i) Q verifies ψ and (ii) Q verifies ψ implies P verifies ϕ. The main advantage of this approach is that (i) can be verified using standard verification tools, whereas (ii) is proved generically for the method used for constructing Q, for instance self-composition [4], [5] and product programs [6], [7]. Another approach to verify relational properties is to use relational Hoare logic [8] or specialized logics that target specific properties [9]. While both approaches have been applied successfully in several use cases, they suffer from fundamental limitations: (i) they are typically not efficient enough to scale to large programs and (ii) they are only partly automated and tailored to specific properties.

Contributions. In this paper, we develop a new approach based on reduction to first-order reasoning, with the intent of reconciling expressiveness and automation.

(1) We introduce and formally characterize *trace logic* $\mathcal{L}$, an instance of many-sorted first-order logic with equality, which allows expressing properties over program locations, loop iterations, and computation traces (Section IV).

```
 func main()
{
    const Int[] a;
    const Int alength;

    Int i = 0;
    Int hw = 0;

    while (i < alength)
    {
       hw = hw + a[i];
       i = i + 1;
    }
  }
```

Fig. 1: Motivating example.

(2) We *encode the semantics of programs as well as relational program properties in* $\mathcal{L}$ (Section IV). Specifically, given a program P and a relational property F, we construct a first-order formula ξ in $\mathcal{L}$ such that validity of ξ entails that P satisfies F. Note that this semantic characterization stands in contrast with methods based on product programs, Hoare logics, and relational Hoare logics, where verification is syntax-directed.

(3) We show that *relational properties*, such as non-interference, can naturally be *encoded in trace logic* (Section V).

(4) We *implemented our approach in the* RAPID *tool*, which relies on the first-order theorem prover Vampire [10]. We conducted experiments on security-relevant hyperproperties, such as non-interference and sensitivity. Our results show that RAPID is more expressive than state-of-the-art non-interference verification tools and that Vampire is better suited to the verification of security-relevant hyperproperties than state-of-the-art SMT-solvers like Z3 and CVC4.

II. MOTIVATING EXAMPLE

We motivate our work with the simple program of Figure 1. This program iterates over an integer-valued array `a` and stores in the variable `hw` the sum of array elements. If `a` is a bitstring, then this program leaks the so-called Hamming weight of `a` in the variable `hw`. Our aim is to prove the following *relational property* over two arbitrary computation traces t_1 and t_2 of Figure 1: if the elements of the array variable `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

$$\begin{aligned}&\forall k_{\mathbb{I}}.\Big(\big(\forall pos_{\mathbb{I}}.((pos \not\simeq k \land pos \not\simeq k+1) \rightarrow \\ &a(pos, t_1) \simeq a(pos, t_2)) \ \land \ a(k, t_1) \simeq a(k+1, t_2) \\ &\land a(k, t_2) \simeq a(k+1, t_1) \land 0 \leq k+1 < alength\big) \\ &\quad \rightarrow hw(end, t_1) \simeq hw(end, t_2)\Big),\end{aligned} \tag{1}$$

where $k_{\mathbb{I}}$ and $pos_{\mathbb{I}}$ respectively specify that k and pos are of sort integer $\mathbb{I}$. Further, $a(pos, t_i)$ denotes the value of the element at position pos of a in trace t_i, whereas *end* refers to the last program location of Figure 1 (that is, line 14).

Property (1) is challenging to verify, since it requires theory-specific reasoning over integers and it involves alternation of quantifiers, as the length of the array a is unbounded and the k-th position (corresponding to the swap) is arbitrary. To understand the difficulty in automating such kind of reasoning, let us first illustrate how humans would naturally prove property (1). First, split the iterations of the loop of Figure 1 into three intervals: (i) The interval from the first iteration of the loop to the iteration where i has value k, (ii) the interval from the iteration where i has value k to the iteration where i has value $k+2$, and (iii) the interval from the iteration where i has value $k+2$ to the last iteration of the loop. Next, for each of the intervals above, one proves that the equality of the value of hw in traces t_1 and t_2 is *preserved*; that is, if hw has the same value in t_1 and t_2 at the beginning of the interval, then hw also has the same value in t_1 and t_2 at the end of the interval. In particular, for the first and third intervals one uses *inductive reasoning*, to conclude the *preservation* of the equality across the whole interval from the step-wise preservation in the interval of the equality of the value hw in traces t_1 and t_2. Further, for the second interval, one uses commutativity of addition to prove that the value of hw in traces t_1 and t_2 is preserved. By combining that the values of hw in traces t_1 and t_2 are preserved in each of the three intervals, one finally concludes that property (1) is valid.

While the above proof might be natural for humans, it is challenging for automated reasoners for the following reasons: (i) one needs to express and relate different iterations in the execution of the loop in Figure 1 and use these iterations to split the reasoning about loop intervals; (ii) one needs to automatically synthesize the loop intervals whose boundaries depend on values of program variables; and (iii) one needs to combine theory-specific reasoning with induction for proving quantified properties, possibly with alternations of quantifiers. In our work we address these challenges: we introduce *trace logic*, allowing us to express and automatically prove relational properties, including property (1). The key advantages of trace logic are as follows.

(i) In trace logic, program variables are encoded as unary and binary functions over program execution timepoints. This way, we can precisely express the value of each program variable at any program execution timepoint, without introducing abstractions. For Figure 1, for example, we write $hw(end, t_1)$ to denote to the value of hw in trace t_1 at timepoint *end*.

(ii) Trace logic further allows arbitrary quantification over iterations and values of program variables. In particular, we can express and reason about iterations that depend on (possibly non-ground) expressions involving program variables. We use superposition-based first-order reasoning to automate static analysis with trace logic and derive first-order properties about loop iterations, possibly with quantifier alternations. For Figure 1, we generate for example the property $\exists it_{\mathbb{N}}.\big(it < n_9 \land i(l_9(it), t_1) \simeq k\big)$, where l_9 denotes the location where the loop condition is tested and n_9 denotes the first iteration of the loop upon which the loop condition does not hold anymore.

(iii) We guide superposition reasoning in trace logic by using a set of lemmas statically inferred from the program semantics. These lemmas express inductive properties about the program behavior. To illustrate such lemmas, we first introduce the following notation. For an arbitrary program variable v, let $Eq_v(it)$ denote that v has the same value in both traces at iteration it of the loop. For example, for every program variable v of Figure 1, we introduce the following definition:

$$Eq_v(it) := v(l_9(it), t_1) \simeq v(l_9(it), t_2).$$

In particular, for variable hw, we introduce:

$$Eq_{hw}(it) := hw(l_9(it), t_1) \simeq hw(l_9(it), t_2).$$

We then derive the following inductive lemma for each program variable v:

$$\begin{aligned}\forall itB_{\mathbb{N}}.\Big(\big(&Eq_v(\mathtt{0}) \land \forall it_{\mathbb{N}}.((it < itB \land Eq_v(it)) \\ &\rightarrow Eq_v(\mathtt{succ}(it)))\big) \rightarrow Eq_v(itB)\Big),\end{aligned} \tag{2}$$

where $itB_{\mathbb{N}}$ and $it_{\mathbb{N}}$ denote iterations itB, it and $\mathtt{succ}(it)$ denotes the successor of it. Lemma (2) asserts that if v has the same value in traces t_1 and t_2 at the beginning of the loop (that is, at iteration 0) and if the values of v are step-wise equal in traces t_1 and t_2 up to an arbitrary iteration itB, then the values of v are equal in traces t_1 and t_2 at iteration itB (and hence the values of v are preserved in t_1 and t_2 for the entire interval up to itB). For Figure 1, we generate lemma (2) for hw as:

$$\begin{aligned}\forall itB_{\mathbb{N}}.\Big(\big(&Eq_{hw}(\mathtt{0}) \land \forall it_{\mathbb{N}}.((it < itB \land Eq_{hw}(it)) \\ &\rightarrow Eq_{hw}(\mathtt{succ}(it)))\big) \rightarrow Eq_{hw}(itB)\Big).\end{aligned} \tag{3}$$

Note that lemma (2), and in particular lemma (3) for hw, is crucial for proving that the values of hw in traces t_1 and t_2 are the same up to iteration k, as considered in the relational property of (1). With this lemma at hand, we automatically prove property (1) of Figure 1, using superposition reasoning in trace logic.

III. Preliminaries

This section fixes our terminology and programming model.

A. First-order logic

We consider standard many-sorted first-order logic with equality, where equality is denoted by $\simeq$. We allow all standard boolean connectives and quantifiers in the language and write $s \not\simeq t$ instead of $\neg(s \simeq t)$, for two arbitrary first-order terms s and t. A *signature* is any finite set of symbols. We consider equality $\simeq$ as part of the language; hence, $\simeq$ is not a symbol. We write $F_1, \ldots, F_n \vDash F$ to denote that the formula $F_1 \wedge \ldots \wedge F_n \rightarrow F$ is a tautology. In particular, we write $\vDash F$, if F is valid.

By a *first-order theory*, or simply just *theory*, we mean the set of all formulas valid on a class of first-order structures. When we discuss a theory, we call symbols occurring in the signature of the theory *interpreted*, and all other symbols *uninterpreted*. In our work, we consider the combination (union) $\mathbb{N} \cup \mathbb{I}$ of the theory $\mathbb{N}$ of natural numbers and the one $\mathbb{I}$ of integers. The signature of $\mathbb{N}$ consists of standard symbols $\mathtt{0}$, $\mathtt{succ}$, $\mathtt{pred}$ and $<$, respectively interpreted as *zero*, *successor*, *predecessor* and *less*. Note that $\mathbb{N}$ does not contain interpreted symbols for (arbitrary) addition and multiplication. We use the theory $\mathbb{N}$ to represent and reason about loop iterations (see Section IV). The signature of $\mathbb{I}$ consists of the standard integer constants $0, 1, 2, \ldots$ and integer operators $+$, $*$ and $<$. We use the theory $\mathbb{I}$ to represent and reason about integer-valued program variables (see Section IV). Additionally we use two (uninterpreted) sorts as two sets of uninterpreted symbols: (i) the sort Timepoint, written as $\mathbb{L}$, for denoting (unique) timepoints in the execution of the program and (ii) the sort Trace, written as $\mathbb{T}$, for denoting computation traces of a program.

Given a logical variable x and sort S, we write x_S to denote that the sort of x is S. We use standard first-order interpretations/models modulo a theory T, for example modulo $\mathbb{N} \cup \mathbb{I}$. We write $\vDash_T F$ to denote that F holds in all models of T (and hence valid). If I is a model of T, we write $I \vDash_T F$ if F holds in the interpretation I.

B. Programming Model $\mathcal{W}$

We consider programs written in a standard while-like programming language, denoted as $\mathcal{W}$, with mutable and constant integer- and integer-array-variables. The language $\mathcal{W}$ includes standard side-effect free expressions over booleans and integers. Each program in $\mathcal{W}$ consists of a single top-level function `main`, with arbitrary nestings of if-then-else and while-statements. For simplicity, whenever we refer to loops, we mean while-loops. For each statement $\mathtt{s}$, we refer to while-statements in which $\mathtt{s}$ is nested in as *enclosing loops* of $\mathtt{s}$. The semantics of $\mathcal{W}$ is formalized in Section IV-C.

IV. Trace Logic

We now introduce the concept of *trace logic* for expressing both the semantics and (relational) properties of $\mathcal{W}$-programs.

A. Locations and Timepoints

We consider a program in $\mathcal{W}$ as a set of locations, where each location intuitively corresponds to a point in the program at which an interpreter can stop. That is, for each program statement $\mathtt{s}$, we introduce a program location l_s. We denote by l_{end} the location corresponding to the end of the program.

As program locations can be revisited during program executions, for example due to the presence of loops, we model locations as follows. For each location l_s corresponding to a program statement $\mathtt{s}$, we introduce a function symbol l_s with target sort $\mathbb{L}$ in our language, denoting the timepoint where the interpreter visits the location. For each enclosing loop of the statement $\mathtt{s}$, the function symbol l_s has an argument of type $\mathbb{N}$; this way, we distinguish between different iterations of the enclosing loop of $\mathtt{s}$. We denote the set of all such function-symbols l_s as Sig_{Tp}. When $\mathtt{s}$ is a loop, we additionally include a function symbol n_s with target sort $\mathbb{N}$ and an argument of sort $\mathbb{N}$ for each enclosing loop of $\mathtt{s}$. This way, n_s denotes the iteration in which $\mathtt{s}$ terminates for given iterations of the enclosing loops of $\mathtt{s}$. We denote the set of all such function symbols n_s as Sig_n.

Example 1: Consider Figure 1. We abbreviate each statement $\mathtt{s}$ by the line number of the first line of $\mathtt{s}$. We use l_6 to refer to the timepoint corresponding to the first assignment of $\mathtt{i}$ in the program. We denote by $l_9(\mathtt{0})$ and $l_9(n_9)$ the timepoints corresponding to evaluating the loop condition in the first and, respectively, last loop iteration. Further, we write $l_{11}(it)$ and $l_{11}(\mathtt{succ}(\mathtt{0}))$ for the timepoint corresponding to the beginning of the loop body in the it-th and, respectively, second iteration of the loop. Note that $\mathtt{succ}(\mathtt{0})$ is a term algebra expression of $\mathbb{N}$. □

For simplicity, let us define terms over the most commonly used timepoints. First, define it^s to be a function, which returns for each while-statement $\mathtt{s}$ a unique variable of sort $\mathbb{N}$. Second, let $\mathtt{s}$ be a statement, let $w_1, \ldots, w_k$ be the enclosing loops of $\mathtt{s}$ and let it be an arbitrary term of sort $\mathbb{N}$.

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

Third, let $\mathtt{s}$ be an arbitrary statement. We refer to the timepoint where the execution of $\mathtt{s}$ has started (parameterized by the enclosing iterators) by

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

Fourth, for an arbitrary statement $\mathtt{s}$, let $end_{\mathtt{s}}$ denote the timepoint which follows immediately after $\mathtt{s}$ has been evaluated completely (including the evaluation of substatements of $\mathtt{s}$):

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

B. Program Variables and Expressions

In our setting, we reason about program behavior by expressing properties over program variables `v`. To do so, we capture the value of program variables `v` at timepoints (from $\mathbb{L}$) in arbitary program execution traces (from $\mathbb{T}$). Hence, we model program variables `v` as functions $v : (\mathbb{L} \times \mathbb{T}) \mapsto \mathbb{I}$, where $v(tp, tr)$ gives the value of `v` at timepoint tp, in trace tr. If the program variable `v` is an array, we add an additional argument of sort $\mathbb{I}$, which corresponds to the position at which the array is accessed. We denote by S_V the set of such introduced function symbols denoting program variables. We finally model arithmetic constants and program expressions using integer functions.

Note that our setting can be simplified for (i) non-mutable variables – in this case we omit the timepoint argument in the function representation of the variable; (ii) for non-relational properties about programs – in this case, we only focus on one computation trace and hence the trace argument in the function from S_V can be omitted.

Example 2: Consider again Figure 1. By $i(l_6, tr)$ we refer to the value of program variable `i` in trace tr at the moment before `i` is first assigned. We use $alength(tr)$ to refer to the value of variable `alength` in trace tr. As `a` is unchanged in the program, we write $a(i(l_{11}(it), tr), tr)$ for the value of array `a` in trace tr at position pos, where pos is the value of `i` in trace tr at timepoint $l_{11}(it)$. In case `a` would have changed during the loop, we would have written $a(l_{11}(it), i(l_{11}(it), tr), tr)$ instead. We denote by $i(l_{12}(it), tr) + 1$ the value of the expression `i+1` in trace tr at timepoint $l_{12}(it)$. □

Consider now an arbitrary program expression `e`. We write $[\![\mathtt{e}]\!](tp, tr)$ to denote the value of `e` at timepoint tp, in trace tr. With these notations at hand, we introduce two definitions expressing properties about values of expressions `e` at arbitrary timepoints and traces. Consider now $v \in S_V$, that is a function denoting a program variable `v`, and let tp_1, tp_2 denote two timepoints. We define: $Eq(v, tp_1, tp_2) :=$

$$\begin{cases} \forall pos_{\mathbb{I}}.\ \ v(tp_1, pos, tr) \simeq v(tp_2, pos, tr), & \text{if } \mathtt{v} \text{ is array} \\ \qquad v(tp_1, tr) \simeq v(tp_2, tr), & \text{otherwise} \end{cases} \tag{4}$$

That is, $Eq(v, tp_1, tp_2)$ in (4) states that the program variable `v` has the same values at tp_1 and tp_2. We also define:

$$EqAll(tp_1, tp_2) := \bigwedge_{v \in S_V} Eq(v, tp_1, tp_2), \tag{5}$$

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

C. Semantics of $\mathcal{W}$

We now describe the semantics of $\mathcal{W}$ expressed in our *trace logic* $\mathcal{L}$. To do so, we state *trace axioms of* $\mathcal{L}$ capturing the behavior of possible program computation traces and then define $\mathcal{L}$.

In what follows, we consider an arbitrary but fixed program P in $\mathcal{W}$, and give all definitions relative to P. Note that our semantics defines arbitrary executions, which are modeled by a free variable tr of sort $\mathbb{T}$.

a) Main-function: Let $\mathtt{s_1}, \dots, \mathtt{s_k}$ be statements and P be a program with top-level function `func main {s1;...;sk}`. The semantics of P is defined by the conjunction of the semantics of the statements $\mathtt{s_i}$ in the top-level function and is the same for each trace. That is:

$$[\![P]\!] := \bigwedge_{i=1}^{k} [\![\mathtt{s_i}]\!]. \tag{6}$$

The semantics of P is then defined by structural induction, by asserting *trace axioms* for each program statement `s`, as follows.

b) Skip: Let `s` be a statement `skip`. The evaluation of s has no effect on the value of the program variables. Hence:

$$[\![\mathtt{s}]\!] := \bigwedge_{v \in S_V} Eq(v, end_{\mathtt{s}}, tp_{\mathtt{s}}) \tag{7}$$

c) Integer assignments: Let `s` be an assignment `v = e`, where `v` is an integer program variable and `e` is an expression. We reason as follows. The assignment `s` is evaluated in one step. After the evaluation of `s`, the variable `v` has the same value as `e` before the evaluation, and all other variables remain unchanged. Hence:

$$[\![\mathtt{s}]\!] := v(end_{\mathtt{s}}) \simeq [\![\mathtt{e}]\!](tp_{\mathtt{s}}, tr) \wedge \bigwedge_{v' \in S_V \setminus \{v\}} Eq(v', end_{\mathtt{s}}, tp_{\mathtt{s}}) \tag{8}$$

d) Array assignments: Let `s` be an assignment $\mathtt{a[e_1]} = \mathtt{e_2}$, where `a` is an array variable and $\mathtt{e_1}, \mathtt{e_2}$ are expressions. We consider that the assignment is evaluated in one step. After the evaluation of `s`, the array `a` has the same value as before the evaluation, except for the position pos corresponding to the value of $\mathtt{e_1}$ before the evaluation, where the array now has the value of $\mathtt{e_2}$ before the evaluation. All other program variables remain unchanged and we have:

$$\begin{aligned} [\![\mathtt{s}]\!] := \quad & \forall pos_{\mathbb{I}}.(pos \not\simeq e_1(tp_{\mathtt{s}}, tr) \rightarrow \\ & \quad a(end_{\mathtt{s}}, pos, tr) \simeq a(tp_{\mathtt{s}}, pos, tr)) & \text{(9a)} \\ \wedge \quad & a(end_{\mathtt{s}}, e_1(tp_{\mathtt{s}}, tr)) \simeq e_2(tp_{\mathtt{s}}, tr) & \text{(9b)} \\ \wedge \quad & \bigwedge_{v \in S_V \setminus \{a\}} Eq(v, end_{\mathtt{s}}, tp_{\mathtt{s}}) & \text{(9c)} \end{aligned}$$

e) Conditional if-then-else Statements: Let `s` be the statement: `if(Cond){s1;...;sk} else {s1';...;sk''}`. The semantics of `s` is defined by the following two properties: (i) entering the if-branch and/or entering the else-branch does not change the values of the variables, (ii) the evaluation in the branches proceeds according to the semantics of the statements in each of the branches. Thus:

$$\begin{aligned} [\![\mathtt{s}]\!] := \quad & [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}) \rightarrow EqAll(start_{\mathtt{s_1}}, tp_{\mathtt{s}}) & \text{(10a)} \\ \wedge \quad & \neg[\![\mathtt{Cond}]\!](tp_{\mathtt{s}}) \rightarrow EqAll(start'_{\mathtt{s_1}}, tp_{\mathtt{s}}) & \text{(10b)} \\ \wedge \quad & [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}) \rightarrow [\![\mathtt{s_1}]\!] \wedge \dots \wedge [\![\mathtt{s_k}]\!] & \text{(10c)} \\ \wedge \quad & \neg[\![\mathtt{Cond}]\!](tp_{\mathtt{s}}) \rightarrow [\![\mathtt{s_1}']\!] \wedge \dots \wedge [\![\mathtt{s_{k'}}']\!] & \text{(10d)} \end{aligned}$$

f) While-Loops: Let $\mathtt{s}$ be the while-statement `while(Cond){s_1;...;s_k}`. We refer to `Cond` as the *loop condition*. We use the following four properties to defined the semantics of $\mathtt{s}$: (i) the iteration $lastIt_{\mathtt{s}}$ is the first iteration where the loop condition does not hold, (ii) entering the loop body does not change the values of the variables, (iii) the evaluation in the body proceeds according to the semantics of the statements in the body, (iv) the values of the variables at the end of evaluating $\mathtt{s}$ are the same as the variable values at the loop condition location in iteration $lastIt(\mathtt{s})$. We then have:

$$\llbracket \mathtt{s} \rrbracket :=$$

$$\forall it^s_{\mathbb{N}}.\ (it^s < lastIt_{\mathtt{s}} \rightarrow \llbracket \mathtt{Cond} \rrbracket(tp_{\mathtt{s}}(it^s))) \tag{11a}$$

$$\wedge\quad \neg\llbracket \mathtt{Cond} \rrbracket(tp(lastIt_{\mathtt{s}})) \tag{11b}$$

$$\wedge\quad \forall it^s_{\mathbb{N}}.\ (it^s < lastIt_{\mathtt{s}} \rightarrow EqAll(start_{\mathtt{s}_1}, tp_{\mathtt{s}}(it^s)) \tag{11c}$$

$$\wedge\quad \forall it^s_{\mathbb{N}}.\ (it^s < lastIt_{\mathtt{s}} \rightarrow (\llbracket \mathtt{s_1} \rrbracket \wedge \cdots \wedge \llbracket \mathtt{s_k} \rrbracket) \tag{11d}$$

$$\wedge\quad EqAll(end_{\mathtt{s}}, tp_s(lastIt_{\mathtt{s}})) \tag{11e}$$

D. Trace Logic $\mathcal{L}$

We now have all ingredients to define our *trace logic* $\mathcal{L}$, allowing us to reason about both relational and non-relational properties of programs.

Let S_{Tr} be a set $\{t_1, t_2, \dots\}$ of nullary function symbols of sort $\mathbb{T}$. Intuitively, these symbols denote traces and allow us to express relational properties. The signature of $\mathcal{L}$ contains the symbols of the theories $\mathbb{N}$ and $\mathbb{I}$ together with symbols introduced in Section IV-A-IV-B, that is symbols denoting timepoints, last iterations in loops, program variables and traces. Formally,

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

Recall that the semantics of $\mathcal{W}$ is defined by the trace axioms (7)-(11). By extending standard small-step operational semantics with timepoints and traces, we obtain the small-step semantics of $\mathcal{W}$. 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 refer to [11] for more details. We then introduce $\mathcal{W}$-soundness defining the soundness of the semantics of $\mathcal{W}$, as follows:

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 (7)-(11) are $\mathcal{W}$-sound.

As a consequence, the semantics of any terminating program p expressed in $\mathcal{L}$, as defined in (6), is $\mathcal{W}$-sound.

E. 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 $\llbracket P \rrbracket$ in $\mathcal{L}$, as discussed in Section IV-C;
2) We prove the partial correctness of P with respect to F; that is, we prove

$$\llbracket P \rrbracket \ \vDash_{\mathbb{N}\cup\mathbb{I}} \ F.$$

In what follows, we first discuss (relational) properties F expressed in $\mathcal{L}$ (Section V) and then focus on proving partial correctness using $\mathcal{L}$ (Section VI).

V. Hyperproperties in Trace Logic

We demonstrate the expressiveness of trace logic $\mathcal{L}$ by encoding non-interference [12] and sensitivity [13], two fundamental security properties. For space restriction, we only exemplify non-interference and we refer to [11] for reasoning about sensitivity in trace logic $\mathcal{L}$. This secition also showcases the generic lemmas, similar to property (2), introduced by our work to automate the verification of hyperproperties. The examples considered in this section are deemed as insecure by existing syntax-driven, non-interference verification techniques, such as [12], [14].

a) Non-interference: Non-interference [1] is a security property that prevents information flow from confidential data to public channels. It is a so-called 2-safety property expressing that, given two runs of a program containing high and low confidentiality variables, denoted by H and L respectively, if the input for all L variables is the same in both runs, the output of the computation should result in the same values for L variables in both traces regardless of the initial value of any H variable. Intuitively, this means that no private input leaks to any public sink. In what follows, we let `lo` denote an L variable and `hi` an H variable.

We formalize non-interference in trace logic $\mathcal{L}$ as follows. Let l_0 denote the first timepoint of the execution and let $EqTr(v, tp)$ denote that v has the same value(s) in both traces at timepoint tp, that is:

$$EqTr(v, tp) :=$$

$$\begin{cases} \forall pos_{\mathbb{I}}.v(tp, pos, t_1) \simeq v(tp, pos, t_2)) & \text{if } v \text{ is mutable array} \\ \forall pos_{\mathbb{I}}.v(pos, t_1) \simeq v(pos, t_2)) & \text{if } v \text{ is constant array} \\ v(tp, t_1) \simeq v(tp, t_2)) & \text{if } v \text{ is mutable var.} \\ v(t_1) \simeq v(t_2) & \text{if } v \text{ is constant var.} \end{cases}$$

We then express non-interference as:

$$(\bigwedge_{v \in L} EqTr(v, l_0)) \rightarrow (\bigwedge_{v \in L} EqTr(v, l_{end})). \tag{12}$$

Example 3: Consider the program illustrated in Figure 2a, which branches on an H guard. In the two branches, however, the L variable is updated in the same way, thereby not leaking anything about the guard. The non-interference property for this program is a special instance of property (12), as follows:

$$EqTr(lo, l_0) \rightarrow EqTr(lo, l_{end}). \tag{13}$$

By adjusting superposition reasoning to trace logic $\mathcal{L}$ (see Section VI), we can automatically verify the property above. Traditional information-flow type systems [12] would however

```
1  func main()
2  {
3      const 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 }
```

(a) Branching on a *high* variable.

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

(b) Explicit flow.

Fig. 2: Examples with non-interference behaviour.

fail to prove this program secure, as they prevent any branching on H guards. More permissive static analysis techniques based on program dependency graphs, such as JOANA [14], would also classify this program as insecure. □

Let us now focus on another interesting security example.

Example 4: Figure 2b models an interactive program outputting on a public channel. The array variable output $\in L$ models the number and content of these outputs, which is determined by the loop. At a first glance, this program might look insecure because of the explicit flow at l_{11}. Furthermore, the number of outputs, as well as their content, could also leak information about the secret. Indeed, value-insensitive information-flow type systems [12] would consider this program to be insecure. In this specific case, however, the H variable in the loop guard is reset with an L input, and the program satisfies non-interference. As our semantic reasoning in trace logic $\mathcal{L}$ is value sensitive, our work correctly validates Figure 2b. proving it to be secure. Specifically, we prove the following property, stating that if all variables in L are equal at the beginning of the execution, then the values of the output array are equal after the execution:

$$\begin{aligned}&(EqTr(k,l_{11}) \wedge EqTr(lo,l_{11}) \wedge EqTr(output,l_{11}))\\ &\to EqTr(output,l_{end})\end{aligned} \tag{14}$$

□

Our framework generates and relies upon a set of generic *trace lemmas* for hyperproperties, similar to lemma (2). We now illustrate two further such lemmas.

b) Simultaneous-loop-termination: Our semantic formalization of $\mathcal{W}$ in trace logic $\mathcal{L}$ defines $n_s(t_1)$ to be the smallest iteration, in which the loop condition does not hold in trace t_1. Due to well-founded orderings over naturals, there can only be one iteration with such a property. Thus, if we can conclude this property for any other trace, say t_2, then it must be the case that $n_s(t_2) \simeq n_s(t_1)$. In our work we therefore generate and use the following trace lemma in $\mathcal{L}$ (for simplicity, we omit the enclosing iterators):

$$\begin{aligned}&\Big(\forall it.\big(it < n_s(t_1) \to [\![\mathtt{Cond}]\!](l_s(it),t_2)\big)\wedge\\ &\neg[\![\mathtt{Cond}]\!](l_s(n_s(t_1)),t_2)\Big) \to n_s(t_2) \simeq n_s(t_1)\end{aligned} \tag{15}$$

Property (15) is essential to prove that the loops in both traces have the same last iteration, and therefore terminate after the same number of iterations.

c) Equality-preservation-arrays: For an array variable a and loop location l, let $Eq_a(it, pos)$ denote that a at position *pos* has the same value in both traces at iteration *it* of the loop:

$$Eq_a(it,pos) := a(l(it),pos,t_1) \simeq a(l(it),pos,t_2).$$

The following lemma over array variables is similar to the equality-preservation-lemma (2):

$$\begin{aligned}&\forall pos_{\mathbb{I}}.\forall it'_{\mathbb{N}}.\\ &\quad\Big(\big(Eq_a(0,pos) \wedge \forall it_{\mathbb{N}}.((it < it' \wedge Eq_a(it,pos))\\ &\qquad \to Eq_a(\mathtt{succ}(it),pos))\big) \to Eq_a(it',pos)\Big)\end{aligned} \tag{16}$$

We conclude by emphasizing that trace lemmas, such as (15) and (16), are expressed in trace logic $\mathcal{L}$ and automatically generated by our approach.

VI. Implementation and Experiments

A. Implementation

We implemented our approach in the tool RAPID[1], which consists of nearly 13,000 lines of C++ code. RAPID takes as input a program written in $\mathcal{W}$ and a property expressed in trace logic $\mathcal{L}$. It then generates axioms written in trace logic $\mathcal{L}$ corresponding to the semantics of the program and outputs both the axioms and the property in the SMT-LIB syntax [15]. The produced SMT-LIB encoding is further passed within RAPID to the first-order theorem prover VAMPIRE for proving validity of the property (i.e. partial correctness). VAMPIRE searches for a refutation of the desired property by saturating the provided encoding with respect to a set of inference rules such as resolution and superposition [10].

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

a) Inductive Reasoning: Trace logic $\mathcal{L}$ encodes loop-iterations using counters of sort $\mathbb{N}$. Hence, there are consequences of the semantics which can only be derived using inductive reasoning. Automating induction is however challenging: state-of-the-art SMT solvers and theorem-provers are not able to automatically infer and prove most (inductive) consequences needed by RAPID. In order to address this problem, (i) we identified some of the most important applications of induction that are useful for many programs and (ii) formulated the corresponding inductive properties in trace logic as *trace lemmas*. Some of these lemmas are described in Section II and Section V. Each trace lemma is logically implied by standard induction axioms of natural numbers and the semantics of the program. RAPID generates trace lemmas for each variable and each loop of the program and adds them as axioms to its SMT-LIB output.

b) Theory Reasoning: Reasoning with theories in the presence of quantifiers is yet another challenge for automated reasoners, and hence for VAMPIRE. Different theory encodings lead to very different results. In RAPID, we model integers using the built-in support for integers in VAMPIRE. We experimented with various sound but incomplete axiomatization of integers. We used VAMPIRE with all its built-in theory axioms (option `-tha on`, default), as well as with a partial, but most relevant set of theory axioms (option `-tha some`) which we extended with specific integer theory axioms. Natural numbers are modeled in RAPID as a term algebra $(\mathtt{0}, \mathtt{succ}, \mathtt{pred})$, for which efficient reasoning engines already exist [16]. In order to express the ordering of natural numbers, we manually add the symbol $<$, together with an (incomplete) axiomatization. In RAPID, we also experimented with clause splitting by calling VAMPIRE both with and without its AVATAR framework [17] (options `-av on/off`, with `on` as default).

B. Benchmarks and Experimental Results

To compensate the lack of general benchmarks for first-order hyperproperties, we collected a set of 27 verification problems for evaluating our work in RAPID. Our benchmarks describe 2-safety properties relevant in the security domain, such as non-interference and sensitivity. The individual benchmark programs consist of up to 50 lines of code each.

RAPID produced the SMT-LIB-encodings for each benchmark in less than a second. These encodings were passed to VAMPIRE, as well as to the SMT solvers Z3 [18] and CVC4 [19] for comparison purposes, to establish the correctness of the input property. We ran each prover with a 60 seconds time limit. All experiments were carried out on an Intel Core i5 3.1Ghz machine with 16 GB of RAM.

Our experimental results are summarized in Table I. The first four columns report on results by running VAMPIRE on the RAPID output. The columns denoted with `S/F` refer to VAMPIRE options for partial/full theory reasoning (option `-tha some/on`) respectively. `A` refers to the use of the AVATAR (option `-av on`) in conjunction with one of the theory options, hence columns `S+A` and `F+A`. The last two columns of Table I summarize our results of running Z3 and CVC4 on the RAPID output. The rows denoted *Total* VAMPIRE and *Unique* VAMPIRE sum up the total and unique numbers of examples proven with the setting of the corresponding column. Example `4-hw-swap-in-array` in Table I is our running example from Figure 1, whereas the benchmarks `3-ni-high-guard-equal-branches` and `9-ni-equal-output` correspond to Figure 2a and Figure 2b, respectively.

Benchmarks	Vampire				CVC4	Z3
	S	S+A	F	F+A		
1-hw-equal-arrays	✓	✓	-	✓	✓	✓
2-hw-last-position-swapped	-	✓	-	-	✓	✓
3-hw-swap-and-two-arrays	-	✓	-	-	-	-
4-hw-swap-in-array-lemma	-	✓	-	-	-	-
4-hw-swap-in-array-full	-	✓	-	-	-	-
1-ni-assign-to-high	✓	✓	✓	✓	✓	✓
2-ni-branch-on-high-twice	✓	✓	✓	✓	✓	✓
3-ni-high-guard-equal-branches	✓	✓	✓	✓	✓	✓
4-ni-branch-on-high-twice-prop2	✓	✓	-	-	✓	✓
5-ni-temp-impl-flow	-	-	✓	✓	✓	✓
6-ni-branch-assign-equal-val	-	-	✓	✓	✓	✓
7-ni-explicit-flow	✓	✓	✓	✓	✓	✓
8-ni-explicit-flow-while	✓	✓	-	✓	✓	✓
9-ni-equal-output	✓	-	-	-	-	✓
10-ni-rsa-exponentiation	✓	✓	✓	✓	✓	-
1-sens-equal-sums	✓	✓	✓	✓	✓	✓
2-sens-equal-sums-two-arrays	✓	✓	✓	✓	-	-
3-sens-abs-diff-up-to-k	-	-	-	-	✓	✓
4-sens-abs-diff-up-to-k-two-arrays	-	-	-	-	-	-
5-sens-two-arrays-equal-k	✓	✓	✓	✓	-	-
6-sens-diff-up-to-explicit-k	✓	✓	✓	✓	-	-
7-sens-diff-up-to-explicit-k-sum	-	-	✓	✓	-	-
8-sens-explicit-swap	-	-	✓	✓	-	-
9-sens-explicit-swap-prop2	-		✓	✓	-	-
10-sens-equal-k	✓	✓	✓	✓	-	-
11-sens-equal-k-twice	✓	✓	✓	✓	-	-
12-sens-diff-up-to-forall-k	-	-	✓	✓	✓	-
Total VAMPIRE	15	18	17	19		
Unique VAMPIRE	1	4	0	0		
Total	25				14	13

TABLE I: RAPID results with VAMPIRE, Z3 and CVC4.

VAMPIRE proved 25 RAPID encodings out of the 27 benchmark problems. Table I shows that the option `S+A` seems to be the most successful, with four unique benchmarks proven. While two of our benchmarks were not proven by VAMPIRE with our current set of automatically generated RAPID lemmas, these problems could actually be proved by VAMPIRE by using only a subset of trace lemmas, i.e. by removing unnecessary lemmas manually. Improving theory reasoning in VAMPIRE, and in general in superposition proving, would further improve the efficiency of RAPID. In particular, designing better reasoning support for transitive relations like $<_{\mathbb{N}}$ and $<_{\mathbb{I}}$ is an interesting further line of research.

We also compared the performance of VAMPIRE on the RAPID examples to the performance of Z3 and CVC4. Unlike VAMPIRE, Z3 and CVC4 proved only 13 and 14 examples, respectively. Our results thus showcase that superposition reasoning, in particular VAMPIRE, is better suited for proving first-order hyperproperties, as many of these properties involve heavy use of quantifiers, including alternations of quantifiers (such as for example `4-hw-swap-in-array` corresponding to Figure 1). Moreover, RAPID proved security of examples that were classified insecure by existing techniques [12], [14], such as `3-ni-high-guard-equal-branches` and `9-ni-equal-output`.

VII. Related Work

Deductive verification. Most verification approaches use a state-based language to express programs and properties about them, and use invariants to establish program correctness [20]. Such invariants loosely correspond to a fragment of trace logic, where formulas only feature universal quantification over time – but no existential quantification. The lack of existential, and thus alternating, quantification makes these works suitable for automation via SMT-solving [21], [22] and hence applicable for programs where full first-order logic is not needed, for instance programs involving mainly integer variables and function calls. For program properties expressed in full first-order logic, such as over unbounded arrays, existing methods are yet not able to automatically verify program correctness. We argue that the missing expressiveness is the problem here, since one usually needs to be able to express arbitrary dependencies of timepoints and values, if custom code is used to iterate through an array or more generally through a data structure. Our trace logic supports such kind of first-order reasoning.

Our approach to automate induction using trace lemmas is related to template-based invariant generation methods [23], [24]. Our trace lemmas are however more expressive than existing templates and we automatically derive trace lemmas.

Program analysis by first-order reasoning is also studied in [25], where program semantics is expressed in extensions of Hoare Logic with explicit timepoints. Unlike [25], we do not rely on an intermediate program (Hoare) logic, but reason also about relational properties. While [25] can only handle simple loops, our work supports a standard while-language with explicit locations and arbitrary nestings of statements.

First-order reasoning for program analysus is also addressed in [26], by introducing dynamic trace logic: an extension of dynamic logic with modalities for reasoning about traces. A custom sequent calculus is proposed in [26], implying that automating the work would require the design of specialised sequent calculus provers. Unlike [26], our work is fully automated. Further, our work preserves the control-flow structure of programs by introducing function symbols and automates inductive reasoning using trace lemmas.

Relational verification. Verification of relational- and hyperproperties is an active area of research, with applications in programming languages and compilers, security and privacy; see [27] for an overview. Various static analysis techniques have been proposed to analyze non-interference, such as type systems [12] and graph dependency analysis [14]. Type systems proved also effective in the verification of privacy properties for cryptographic protocols [28]–[32]. Relational Hoare logic was introduced in [8] and further extended in [6], [33] for defining product programs to reduce relational verification to standard verification. All these works closely tie verification to the syntactic program structure, thus limiting their applicability and expressiveness. As already argued, our work allows proving security of examples that were so far classified as insecure by some of the aforementioned methods [12], [14]. Recently, [34] encodes relational properties through refinement types in F* [35]. While still being syntax driven, [34] can potentially verify semantic properties by using SMT solving, although this typically requires the manual insertion and proof of program-dependent lemmas, which is not the case for us.

In [36] bounded model checking is proposed for program equivalence. Program equivalence is reduced in [37] to proving a set of Horn clauses, by combining a relational weakest precondition calculus with SMT-based reasoning. However, when addressing programs with different control flow as in [37], user guidance is required for proving program equivalence. Program equivalence is also studied in [38], [39] for proving information flow properties. Unlike these works, we are not limited to SMT solving but automate the verification of relational properties expressed in full first-order theories, possibly with alternations of quantifiers.

Motivated by applications to translation validation, the work of [40] develops powerful techniques for proving correctness of loop transformations. Relational methods for reasoning about program versions and semantic differences are also introduced in [41], [42]. Going beyond relational properties, an SMT-based framework for verifying k-safety properties is introduced in [43] and further extended [44] for proving correctness of 3-way merge. While these works focus on high-level languages, many others consider low-level languages, see [45]–[48] for some exemplary approaches. Further afield, several authors have introduced logics for modelling hyperproperties. Unlike these works, trace logic allows expressing first-order relational properties and automates reasoning about such properties by first-order theorem proving, overcoming thus the SMT-based limitations of quantified reasoning.

Finally, in [49] HyperLTL and HyperCTL* is introduced to model temporal and relational properties properties. However, these logics support only decidable fragments of first-order logic and thus cannot handle relational properties with non-constant function symbols. As such, security and privacy properties over unbounded data structures/uninterpreted functions cannot be encoded or verified.

VIII. Conclusion

We introduced trace logic for automating the verification of relational program properties of imperative programs. We showed that program semantics as well as relational properties can naturally be encoded in trace logic as first-order properties over program locations, loop iterations and computation traces. We combined trace logic with superposition proving and implemented our work in the RAPID tool. While our work already outperforms SMT-based approaches, we are convinced that improving superposition reasoning with both theories and quantifiers would further strengthen the use of trace logic for relational verification.

Acknowledgements.

This work was funded by the ERC Starting Grant 2014 SYMCAR 639270, the ERC Proof of Concept Grant 2018 SYMELS 842066, the Wallenberg Academy Fellowship 2014 TheProSE, the Austrian FWF research projects W1255-N23 and RiSE S11409-N23, the ERC Consolidator Grant 2018 BROWSEC 771527, by the Netidee projects EtherTrust 2158 and PROFET P31621, and by the FFG projects PR4DLT 13808694 and COMET K1 SBA.

REFERENCES

[1] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[2] S. Chaudhuri, S. Gulwani, and R. Lublinerman, "Continuity analysis of programs," in *POPL*, 2010, pp. 57–70.

[3] M. R. Clarkson and F. B. Schneider, "Hyperproperties," in *CSF*, 2008, pp. 51–65.

[4] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure Information Flow by Self-Composition," in *CSFW*, 2004, pp. 100–114.

[5] Á. Darvas, R. Hähnle, and D. Sands, "A Theorem Proving Approach to Analysis of Secure Information Flow," in *SPC*, 2005, pp. 193–209.

[6] G. Barthe, J. M. Crespo, and C. Kunz, "Relational Verification Using Product Programs," in *FM*, 2011, pp. 200–214.

[7] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic Program Alignment for Equivalence Checking," in *PLDI*, 2019, pp. 1027–1040.

[8] N. Benton, "Simple Relational Correctness Proofs for Static Analyses and Program Transformations," in *POPL*, 2004, pp. 14–25.

[9] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A Logic for Information Flow in Object-Oriented Programs," in *POPL*, 2006, pp. 91–102.

[10] L. Kovács and A. Voronkov, "First-Order Theorem Proving and Vampire," in *CAV*, 2013, pp. 1–35.

[11] G. Barthe, R. Eilers, P. Georgiou, B. Gleiss, L. Kovács, and M. Maffei, "Verifying Relational Properties using Trace Logic," *CoRR*, vol. abs/1906.09899, 2019.

[12] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE J. on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[13] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating Noise to Sensitivity in Private Data Analysis," in *Theory of Cryptography Conference*, 2006, pp. 265–284.

[14] J. Graf, M. Hecker, and M. Mohr, "Using Joana for Information Flow Control in Java Programs – A Practical Guide," *Software Engineering 2013 – Workshopband*, 2013.

[15] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, available at `www.SMT-LIB.org`.

[16] L. Kovács, S. Robillard, and A. Voronkov, "Coming to Terms with Quantified Reasoning," in *POPL*, 2017, pp. 260–270.

[17] A. Voronkov, "AVATAR: The Architecture for First-Order Theorem Provers," in *CAV*, 2014, pp. 696–710.

[18] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, 2008, pp. 337–340.

[19] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, 2011, pp. 171–177.

[20] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn Clause Solvers for Program Verification," in *Fields of Logic and Computation II*, 2015, pp. 24–51.

[21] K. Hoder and N. Bjørner, "Generalized Property Directed Reachability," in *SAT*, 2012, pp. 157–171.

[22] A. Gurfinkel, S. Shoham, and Y. Meshman, "SMT-based Verification of Parameterized Systems," in *FSE*, 2016, pp. 338–348.

[23] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear Invariant Generation Using Non-Linear Constraint Solving," in *CAV*, 2003, pp. 420–432.

[24] A. Gupta and A. Rybalchenko, "InvGen: An Efficient Invariant Generator," in *CAV*, 2009, pp. 634–640.

[25] B. Gleiss, L. Kovács, and S. Robillard, "Loop Analysis by Quantification over Iterations," in *LPAR*, 2018, pp. 381–399.

[26] B. Beckert and S. Schlager, "A Sequent Calculus for First-Order Dynamic Logic with Trace Modalities," in *IJCAR*, 2001, pp. 626–641.

[27] B. Beckert and M. Ulbrich, "Trends in Relational Program Verification," in *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, 2018, pp. 41–58.

[28] F. Eigner and M. Maffei, "Differential Privacy by Typing in Security Protocols," in *CSF*, 2013, pp. 272–286.

[29] G. Barthe, C. Fournet, B. Grégoire, P.-Y. Strub, N. Swamy, and S. Zanella-Béguelin, "Probabilistic Relational Verification for Cryptographic Implementations," in *POPL*, 2014, pp. 193–205.

[30] V. Cortier, F. Eigner, S. Kremer, M. Maffei, and C. Wiedling, "Type-Based Verification of Electronic Voting Protocols," in *POST*, 2015, pp. 303–323.

[31] V. Cortier, N. Grimm, J. Lallemand, and M. Maffei, "A Type System for Privacy Properties," in *CCS*, 2017, pp. 409–423.

[32] ——, "Equivalence Properties by Typing in Cryptographic Branching Protocols," in *POST*, 2018, pp. 160–187.

[33] G. Barthe, J. M. Crespo, and C. Kunz, "Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification," in *LFCS*, 2013, pp. 29–43.

[34] N. Grimm, K. Maillard, C. Fournet, C. Hriţcu, M. Maffei, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy, and S. Zanella-Béguelin, "A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations," in *CPP*, 2018, pp. 130–145.

[35] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin, "Dependent Types and Multi-Monadic Effects in F," in *POPL*, 2016, pp. 256–270.

[36] B. Godlin and O. Strichman, "Regression Verification: Proving the Equivalence of Similar Programs," *Softw. Test., Verif. Reliab.*, vol. 23, no. 3, pp. 241–258, 2013.

[37] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating Regression Verification," in *ASE*, 2014, pp. 349–360.

[38] Q. Zhou, D. Heath, and W. Harris, "Completely Automated Equivalence Proofs," *CoRR*, vol. abs/1705.03110, 2017.

[39] H. Kwon, W. Harris, and H. Esmaeilzadeh, "Proving Flow Security of Sequential Logic via Automatically-Synthesized Relational Invariants," in *CSF*, 2017, pp. 420–435.

[40] K. S. Namjoshi and N. Singhania, "Loopy: Programmable and Formally Verified Loop Transformations," in *SAS*, 2016, pp. 383–402.

[41] N. Partush and E. Yahav, "Abstract Semantic Differencing via Speculative Correlation," in *OOPSLA*, 2014, pp. 811–828.

[42] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs," in *CAV*, 2012, pp. 712–717.

[43] M. Sousa and I. Dillig, "Cartesian Hoare Logic for Verifying k-Safety Properties," in *PLDI*, 2016, pp. 57–69.

[44] M. Sousa, I. Dillig, and S. K. Lahiri, "Verified Three-Way Program Merge," in *OOPSLA*, 2018, pp. 165:1–165:29.

[45] E. W. Smith and D. L. Dill, "Automatic Formal Verification of Block Cipher Implementations," in *FMCAD*, 2008, pp. 1–7.

[46] M. Stepp, R. Tate, and S. Lerner, "Equality-Based Translation Validator for LLVM," in *CAV*, 2011, pp. 737–742.

[47] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken, "Data-Driven Equivalence Checking," in *OOPSLA*, 2013, pp. 391–406.

[48] M. Balliu, M. Dam, and R. Guanciale, "Automating Information Flow Analysis of Low Level Code," in *CCS*, 2014, pp. 1080–1091.

[49] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez, "Temporal Logics for Hyperproperties," in *POST*, 2014, pp. 265–284.