A Derivation Framework for Dependent Security Label Inference

PEIXUAN LI and DANFENG ZHANG, The Pennsylvania State University, United States

Dependent security labels (security labels that depend on program states) in various forms have been introduced to express rich information flow policies. They are shown to be essential in the verification of real-world software and hardware systems such as conference management systems, Android Apps, a MIPS processor and a TrustZone-like architecture. However, most work assumes that all (complex) labels are provided manually, which can both be error-prone and time-consuming.

In this paper, we tackle the problem of automatic label inference for static information flow analyses with dependent security labels. In particular, we propose the first general framework to facilitate the design and validation (in terms of soundness and/or completeness) of inference algorithms. The framework models label inference as constraint solving and offers guidelines for sound and/or complete constraint solving. Under the framework, we propose novel constraint solving algorithms that are both sound and complete. Evaluation result on sets of constraints generated from secure and insecure variants of a MIPS processor suggests that the novel algorithms improve the performance of an existing algorithm by orders of magnitude and offers better scalability.

CCS Concepts: • **Security and privacy** → **Information flow control**; • **Software and its engineering** → *Constraint and logic languages*; *Constraints*; Automated static analysis;

Additional Key Words and Phrases: Information Flow Analysis, Security Label Inference, Dependent Types

ACM Reference Format:
Peixuan Li and Danfeng Zhang. 2018. A Derivation Framework for Dependent Security Label Inference. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 115 (November 2018), 26 pages. https://doi.org/10.1145/3276485

1 INTRODUCTION

Information flow control is a promising way of protecting the confidentiality of information that is manipulated by computer systems. Compared with conventional mechanisms such as access control, information flow control provides fine-grained reasoning about information flows, as well as a strong end-to-end security guarantee: secret inputs cannot be inferred by an attacker through the observations of public outputs.

Compared to dynamic enforcements (e.g., [Bell and LaPadula 1973; Fenton 1974; Le Guernic and Jensen 2005; Sabelfeld and Russo 2010; Shroff et al. 2007]), a static enforcement verifies information flow policies at compile time so that all vulnerabilities are detected before program execution. Hence, there is no computation or storage overhead at runtime. However, it is also well-known that classic static information flow analysis sometimes lacks enough expressiveness for real-world applications. This becomes a key barrier to wide adoption of those static methods.

To improve the expressiveness of static information flow analysis, dependent types in various forms have been introduced. For instance, Jif [Myers et al. 2006] and its extensions [Arden et al.

Authors' address: Peixuan Li; Danfeng Zhang, Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA, 16801, United States, {pzl129,zhang}@cse.psu.edu.

2475-1421/2018/11-ART115
https://doi.org/10.1145/3276485

2012; Liu et al. 2009] introduce *dynamic security labels*, security labels that can be manipulated at runtime. More recent works [Chen et al. 2018; Ferraiuolo et al. 2017; Li and Zhang 2017; Lourenço and Caires 2015; Murray et al. 2016; Polikarpova et al. 2018; Zhang et al. 2015] explore the theories and techniques to apply dependent type theory [Martin-Löf 1982] to information flow control. In a nutshell, dependent security labels (or dependent labels in short) are security types that may *depend on concrete program states*. The added expressiveness leads to successful verification of real-world systems, such as a MIPS processor [Zhang et al. 2015], conference management systems [Lourenço and Caires 2015; Polikarpova et al. 2018], a TrustZone-like architecture [Ferraiuolo et al. 2017] and Android Apps [Chen et al. 2018].

However, one big obstacle of verification via those promising dependent security labels is that most work (except [Chen et al. 2018] and [Polikarpova et al. 2018] that support a restricted form of dependent labels) requires programmers to write down all (dependent) labels. However, since the dependent labels usually involve intricate security invariants, providing those labels requires a deep understanding of the program being verified, making it a both time-consuming and error-prone process. Moreover, when the provided labels are incorrect (i.e., program analysis fails with the labels), it is unclear if the program being verified is insecure, or the program can be verified with other correct labels.

For security labels without dependence, classic security type systems (such as Jif [Myers et al. 2006] and FlowCaml [Simonet 2003]) encode the restrictions on security labels into constraints in a finite semi-lattice. Those constraints can be solved by customized solvers (such as the Rehof-Mogensen algorithm [Rehof et al. 1999] and set-constraints solvers [Aiken 1999; Aiken et al. 1994]). However, they cannot handle the infinite search space of dependence required for dependent labels.

In this paper, we introduce the *first general framework* for designing security label inference algorithms and checking their correctness. For generality, we propose a core constraint language that, to the best of our knowledge, can encode *all* static information flow analyses that allow dependent security labels, except that the current encoding for dynamic labels may not be practically efficient (we defer a more detailed discussion to Section 3.3). In particular, the framework models security restrictions on a program as *predicated constraints*, in the form of $P \rightarrow \tau_1 \sqsubseteq \tau_2$, meaning that security label τ_2 must be more restrictive than τ_1 whenever predicate P holds. A key feature of the language, which also makes the inference challenging, is that a solution may have dependence. For example, a constraint $(b = 1 \rightarrow \alpha \sqsubseteq \mathrm{P}) \wedge (b \neq 1 \rightarrow \mathrm{S} \sqsubseteq \alpha)$ has a dependent solution that α is P (public) whenever $b = 1$; α is S (secret) whenever $b \neq 1$. But a solver that does not allow dependent solution (e.g., an SMT solver with theory on semi-lattice) will simply reject the constraint, since neither P nor S (without dependence) is a solution for α.

The framework models security label inference as an iterative process of solving *derivations* (typically, simpler constraints) from the original constraint set. Hence, it allows great flexibility in inference algorithm design. For example, one potential algorithm may simply work on the variant of constraints where all predicates are removed; another potential algorithm may reject a set of constraints whenever a subset of the constraints is found to be unsatisfiable. An algorithm (such as the one in Chen et al. [2018]) may also directly work on an equivalent derivation of the original constraints. More promising are the novel *iterative* algorithms we develop in this paper, which allow early termination without hurting soundness and completeness.

To facilitate the design and validation (in terms of soundness and completeness) of algorithms in the derivation framework, we distill the key properties for making an algorithm sound and/or complete. We show that various algorithms, including an extension to the inference algorithm in existing work [Chen et al. 2018], can be checked under the framework in a straightforward matter. Moreover, we also designed three novel sound and complete algorithms, namely, *early-accept algorithm*, *early-reject algorithm*, and *hybrid algorithm*. Based on a mix of both satisfiable and unsatisfiable constraint sets collected from the verification of an information flow policy on a MIPS processor using SecVerilog [Ferraiuolo et al. 2017; Zhang et al. 2015], we found that the novel algorithms solve predicated constraints faster than an existing algorithm [Chen et al. 2018] by orders of magnitude.

In summary, this paper makes the following contributions:

- We define a core constraint language that, to the best of our knowledge, can encode label inference for *all* static information flow using dependent labels, though at the moment not all encodings are efficient in practice (Section 3).
- We propose the *first general framework* for the design and validation of label inference algorithms for the core constraint language (Section 4). It formalizes the key properties for an inference algorithm to be sound and/or complete.
- Under the framework, we propose three novel inference algorithms that allow early termination in a sound and complete manner (Section 5).
- We implement and evaluate the novel algorithms on a corpus of predicated constraints collected from secure and insecure variants of a MIPS processor (Section 6). Evaluation result suggests that the novel algorithms scale well, and they outperform existing algorithms.

2 BACKGROUND AND OVERVIEW

2.1 Information Flow Analysis and Security Labels

Conventionally, we assume *security levels* (e.g., P for public and S for secret) are associated with information to describe the intended secrecy of the contents. As standard, we assume the security levels form a lattice $\mathcal{L}$ whose partial ordering $\sqsubseteq$ specifies an information flow policy: information flow from level ℓ_1 to ℓ_2 is allowed if and only if $\ell_1 \sqsubseteq \ell_2$. For simplicity, we use two distinguished security levels $\mathsf{P} \sqsubseteq \mathsf{S}$ throughout the paper, and assume variable s is labeled as S, and variable p is labeled as P unless specified otherwise. However, note that our mechanism applies to any general security lattice.

Most information-flow type systems (e.g., [Hunt and Sands 2006; Myers 1999; Pottier and Simonet 2002; Volpano et al. 1996]) associate one security level from the security lattice to each program variable. To ensure information flow security (typically, some variant of the noninterference property [Goguen and Meseguer 1982]), a type system checks security restrictions on information flows in a program. For example, for an assignment $\mathsf{s} := \mathsf{p}$, the restriction is that the security level of s is more restrictive than that of p (i.e., $\mathsf{P} \sqsubseteq \mathsf{S}$). When the security levels of some variables are missing, some systems (e.g., Jif [Myers et al. 2006] and FlowCaml [Simonet 2003]) try to infer security levels for them whenever possible. Since the security levels form a lattice, a typical inference algorithm encodes inference as solving constraints in a semi-lattice, where sound and complete algorithms exist (e.g., [Rehof et al. 1999]).

However, security levels from a security lattice cannot express several demands in real-world applications. First, some applications require value-dependent security policies. For example, a function "getPwd" that takes a user name x should return a password with x's security level attached. Second, analysis precision can be improved with value-dependent labels. For example, consider the program on the left. This is a secure program since the secret value of s never affects the public variable p_2 (the two assignments under "if" statements are never executed together). However, no static security level for y will work, even in a flow-sensitive system, since at the end of first "if" the level of y can be neither P (the flow from s to y is insecure) nor S (the flow from y to p_2 is insecure).

```
y := 0;
if (p₁ < 0) then y := s;
if (p₁ > 0) then x := y;
p₂ := x;
```

Original Constraints:

$$\begin{aligned} \texttt{true} &\rightarrow \mathrm{P} \sqsubseteq \alpha_x \wedge \alpha_z \sqsubseteq \alpha_x \wedge \alpha_x \sqsubseteq \mathrm{P} \\ d > 0 &\rightarrow \mathrm{S} \sqsubseteq \alpha_y \\ \neg(d > 0) &\rightarrow \mathrm{P} \sqsubseteq \alpha_y \\ d < 0 &\rightarrow \alpha_y \sqsubseteq \alpha_x \end{aligned} \tag{2.1}$$

Derivation 1: (a sound derivation)

$$\begin{aligned} \texttt{true} \rightarrow & \mathrm{P} \sqsubseteq \alpha_x \wedge \alpha_z \sqsubseteq \alpha_x \wedge \alpha_x \sqsubseteq \mathrm{P} \\ & \wedge \mathrm{S} \sqsubseteq \alpha_y \wedge \mathrm{P} \sqsubseteq \alpha_y \wedge \alpha_y \sqsubseteq \alpha_x \end{aligned} \tag{2.2}$$

Derivation 2: (a weaker yet sound derivation)

$$\begin{aligned} d > 0 &\rightarrow \mathrm{P} \sqsubseteq \alpha_x \wedge \alpha_z \sqsubseteq \alpha_x \wedge \alpha_x \sqsubseteq \mathrm{P} \wedge \mathrm{S} \sqsubseteq \alpha_y \\ d \leq 0 &\rightarrow \mathrm{P} \sqsubseteq \alpha_x \wedge \alpha_z \sqsubseteq \alpha_x \wedge \alpha_x \sqsubseteq \mathrm{p} \wedge \mathrm{P} \sqsubseteq \alpha_y \wedge \alpha_y \sqsubseteq \alpha_x \end{aligned} \tag{2.3}$$

Derivation 3: (a sound and complete derivation)

$$\begin{aligned} d > 0 &\rightarrow \mathrm{P} \sqsubseteq \alpha_x \wedge \alpha_z \sqsubseteq \alpha_x \wedge \alpha_x \sqsubseteq \mathrm{P} \wedge \mathrm{S} \sqsubseteq \alpha_y \\ d = 0 &\rightarrow \mathrm{P} \sqsubseteq \alpha_x \wedge \alpha_z \sqsubseteq \alpha_x \wedge \alpha_x \sqsubseteq \mathrm{P} \wedge \mathrm{P} \sqsubseteq \alpha_y \\ d < 0 &\rightarrow \mathrm{P} \sqsubseteq \alpha_x \wedge \alpha_z \sqsubseteq \alpha_x \wedge \alpha_x \sqsubseteq \mathrm{P} \wedge \mathrm{P} \sqsubseteq \alpha_y \wedge \alpha_y \sqsubseteq \alpha_x \end{aligned} \tag{2.4}$$

Fig. 1. Constraints and Sound Derivations.

One promising approach for more expressive information flow security is to apply dependent type theory [Martin-Löf 1982] to information flow control [Chen et al. 2018; Ferraiuolo et al. 2017; Li and Zhang 2017; Lourenço and Caires 2015; Murray et al. 2016; Polikarpova et al. 2018; Zhang et al. 2015]. Such systems allow the security level associated with a variable to *depend on the concrete program state*. The extra expressiveness is shown to be useful in the verification of real-world applications, such as a MIPS processor [Zhang et al. 2015], conference management systems [Lourenço and Caires 2015; Polikarpova et al. 2018], a TrustZone-like architecture [Ferraiuolo et al. 2017] and Android Apps [Chen et al. 2018]. For example, an analysis that allows dependent security labels (e.g., [Li and Zhang 2017]) can verify the secure code above if the following dependent label is given *manually* to y: S when $\mathrm{p}_1 < 0$; P when $\mathrm{p}_1 > 0$.

While dependent security labels are very promising in closing the gap between static information flow analysis and real-world applications, one limitation of existing works on dependent security label is that they either support label inference but only in a restricted form [Chen et al. 2018; Polikarpova et al. 2018], or assume that all (complex) security labels are annotated manually by a programmer. In this paper, we tackle the problem of *automatically inferring dependent security labels for general static information flow analysis with an arbitrary security lattice.*

2.2 Overview

To make label inference general, we model information flow restrictions on a program as *predicated constraints*, in the form of $P \rightarrow \tau_1 \sqsubseteq \tau_2$, meaning that the security label τ_2 must be more restrictive than τ_1 when condition P holds. We call the fragment without P (i.e., $\tau_1 \sqsubseteq \tau_2$) a *label constraint.* For example, (2.1)-(2.4) in Figure 1 are four sets of predicated constraints (we defer a more detailed discussion on how the original constraints are generated to Section 3). Intuitively, each constraint can be interpreted as: whenever some abstract event P (e.g., a program execution that satisfies P) happens, the constraint $\tau_1 \sqsubseteq \tau_2$ is required for security. For example, the second constraint in (2.1) requires that α_y (a security label to be inferred) is confidential whenever the condition $d > 0$ holds. In general, the predicate P can be parameterized over a theory on program states, linear inequalities over integers, boolean constraints, or finite sets.

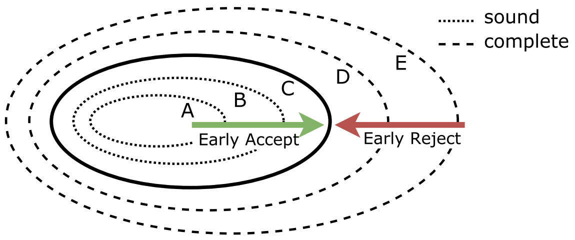

Fig. 2. Sound and Complete Derivations.

What makes it challenging to infer dependent labels is that a label may depend on an *arbitrary* condition, for which we propose a *derivation framework* (Section 4). The key idea is to model inference as an iterative process of transforming the original constraints into a (often) more manageable format. To see why doing so is beneficial, let us consider two simple cases.

- Partition: When a set of constraints have no overlapping predicates, e.g., (2.3), we can easily compute its solution by solving each label constraint without predicate (e.g., via the Rehof-Mogensen algorithm [Rehof et al. 1999]) and merge all local solutions to a global solution.
- Counterexample: If under any predicate, the label constraints are unsatisfiable (i.e., there is no local solution), then there is no global solution.

In general, the original constraints, such as (2.1), can be more challenging: they neither form a partition, nor have an obvious counterexample. Our insight is that an inference algorithm can proceed by transforming the original constraint set into its *derivations*, as illustrated in Figure 2, where each circle represents one such derivation. Of particular interest for constraint solving are three kinds of derivations and constraint solving strategies:

- **Sound Derivation and Early-Accept** A derivation is *sound* if its solution is also a solution of the original constraints. For example, all three derivations: (2.2), (2.3) and (2.4) are sound derivations of (2.1). An *early-accept algorithm* starts with a sound yet simple derivation (e.g., removing all predicates as shown in Derivation (2.2)) and proceeds to the next derivation *only if* the current derivation has no solution. To make such an algorithm sound and complete, an algorithm will intuitively work on a weaker derivation in each iteration until reaching a sound and complete derivation (e.g., Derivation (2.4)), as illustrated by the green arrow in Figure 2.
- **Complete Derivation and Early-Reject** A derivation is *complete* if any solution of the original constraints is a solution of it (i.e., a complete derivation is unsatisfiable implies that the original constraints are unsatisfiable). An *early-reject algorithm* starts with a complete yet simple derivation (e.g., solving constraints under the same predicate) and moves to the next derivation *only if* the current derivation *has a solution*. A sound and complete algorithm can also be designed, as a dual to the early-accept strategy, as illustrated by the red arrow in Figure 2.
- **Equivalent Derivation and One-shot** For both early-accept and early-reject algorithms, their final derivation are both sound and complete. Another possible solving strategy may directly work on an equivalent derivation without looking for the opportunities of early termination. We call such a special case the *one-shot approach*. In fact, one existing algorithm for inferring dependent labels [Chen et al. 2018] is a one-shot algorithm.

An eagle-eyed reader may find that under the derivation model, the main challenge of designing an inference algorithm is to construct derivations and validate their soundness and/or completeness.

Label	$\tau ::= \ell \mid \alpha$, where $\ell \in \mathcal{L}$, $\alpha \in$ **CVars**
Label Constraint	$C ::= \tau_1 \sqsubseteq \tau_2 \mid C_1 \wedge C_2$
Arithmetic Expr	$a ::= x \mid n \mid a \text{ op } a$
Boolean Expr	$b ::= \texttt{true} \mid \texttt{false} \mid a_1 \text{ cop } a_2 \mid b_1 \text{ bop } b_2 \mid \neg b$
Predicates	$P ::= b$
Predicated Constraint	$\mathcal{C} ::= P \to C \mid \mathcal{C} \wedge \mathcal{C}$

Fig. 3. Syntax of Constraints.

The derivation framework offers a couple of core properties of derivations to check against for such purposes (Section 4.2), so that the soundness and/or completeness of derivations can be validated in a simple way.

3 CORE CONSTRAINT LANGUAGE

To enable label inference for various information flow analyses with dependent labels, we formalize a core constraint language with *predicated constraints*.

3.1 Constraint Syntax

The syntax of predicated constraints is given in Figure 3. In this constraint language, a security label is either a known security level ℓ from a lattice $\mathcal{L}$, or an unknown constraint variable α to be solved. A label constraint $\tau_1 \sqsubseteq \tau_2$ denotes the security restriction that label τ_2 must be at least as restrictive as τ_1 (i.e., the lattice allows $\tau_1 \sqsubseteq \tau_2$). Concatenation of two label constraints $C_1 \wedge C_2$ denotes the restriction on both constraints.

A predicated constraint set $\mathcal{C}$ is a concatenation of predicated constraints, each in the form of $P \to C$, where P is a predicate on program state, and C is a label constraint. In general, any practical theory of predicates, such as program logics in decidable theory, program permissions [Chen et al. 2018], are allowed in the core language. To be concrete, we use predicates of boolean program expressions b to express program state in this paper without losing generality: in the inference algorithm we only assume two abstract operations on the predicate logic: $\circ\forall(P)$ (resp. $\circ\exists(P)$) is `true` iff when all free variables are universally (resp. existentially) quantified, P is `true`. In the concrete syntax of P, we use op for arithmetic operations (e.g., $+, -$), cop for binary tests on arithmetic expressions (e.g., $<, \geq$), and bop for boolean operations (e.g., $\wedge, \vee$).

For convenience, we also treat a predicated constraint set $\mathcal{C}$ as a set of constraints in this paper, written as $\{P_1 \to C_1, \ldots, P_n \to C_n\}$. Moreover, we use $\mathbb{P}_{\mathcal{C}}$ to denote the set of predicates in $\mathcal{C}$ (i.e., $\{P_1, \ldots, P_n\}$), and use $\mathbb{C}_{\mathcal{C}}$ to denote the set of label constraints in $\mathcal{C}$ (i.e., $\{C_1, \ldots, C_n\}$).

Consider the sets of predicated constraints in Figure 1, where the predicates over-approximate possible program states at certain program points. The second predicated constraint in (2.1) reads as: whenever the program state satisfies $d > 0$, the restriction that S $\sqsubseteq \alpha_y$ must hold. We will provide more examples that connect predicated constraints to information flow analysis in Section 3.3.

3.2 Constraint Validity and Satisfiability

When a predicated constraint set $\mathcal{C}$ involves no variable, its *validity* is defined in a trivial way: $\mathcal{C}$ is valid iff for any $P \to C$ in $\mathcal{C}$, C is valid (i.e., all label constraints obey the partial ordering in lattice $\mathcal{L}$). Note that when P is `false`, a constraint is vacuously true; hence, such constraints are excluded from further analysis.

Predicated Constraint Set	$pconset ::= C$
Level Solution	$\eta \in \mathbf{CVars} \to \ell$
Type System	$\Gamma \in \mathbf{Vars} \to e$

$$\text{V-LEVEL}\ \frac{}{\langle s, \ell \rangle \Downarrow \ell} \qquad \text{V-VAR}\ \frac{s(\alpha) = \ell}{\langle s, \alpha \rangle \Downarrow \ell} \qquad \text{V-LCON}\ \frac{\langle s, \tau_1 \rangle \Downarrow \ell_1 \quad \langle s, \tau_2 \rangle \Downarrow \ell_2 \quad \ell_1 \sqsubseteq \ell_2}{[\![\tau_1 \sqsubseteq \tau_2]\!]_s} \qquad \text{V-AND}\ \frac{[\![C_1]\!]_s \quad [\![C_2]\!]_s}{[\![C_1 \wedge C_2]\!]_s}$$

$$\text{V-PCON}\ \frac{[\![C_i]\!]_{s_j} \text{ for any } P_j \to s_j \in \kappa \text{ such that } P_i \wedge P_j \text{ is satisfiable}}{\kappa \models \{P_1 \to C_1, \ldots, P_n \to C_n\}}$$

Fig. 4. Correctness of a Solution κ.

When C involves constraint variables, intuitively, C is *satisfiable* if and only if there is a *solution* κ which maps constraint variables to security levels so that constraints after substitution are valid. Since constraints are dependent, it is not surprising that a constraint solution κ also involves predicates, in the form of $\{P_1 \to s_1, \ldots, P_n \to s_n\}$ where s_i is a *label solution* that maps each constraint variable to a concrete security level.[1] For example, a (predicated) solution $(d > 0 \to \{\alpha_x \mapsto \mathrm{P}\}, d \le 0 \to \{\alpha_x \mapsto \mathrm{S}\})$ means that the solution of α_x is P iff $d > 0$ holds.

To simplify technical development in the constraint language theory, we further restrict that all predicates in a solution form a partition, meaning that no two predicates intersect and the union of all predicates is the same as `true` in the predicate logic:

Definition 1 (Predicate Partition). *We say a predicate set $\{P_1, \ldots, P_n\}$ is a partition iff it satisfies both:*

(1) $\bigwedge_{1 \le i < j \le n} \neg(P_i \wedge P_j)$
(2) $\bigvee_{1 \le i \le n} P_i$

Although such a requirement seems to be restrictive at first glance, it actually allows all legal solutions: a "solution" where some predicates overlap either has a conflict (e.g., $(d > 0 \to \{\alpha_x \mapsto \mathrm{P}\}, d > 1 \to \{\alpha_x \mapsto \mathrm{S}\})$) or can be normalized into a solution by merging the shared solution for the intersection. For condition 2), intuitively, a solution should provide an answer for any possible predicate in the constraints.

Next, we formally define the correctness of a predicated solution κ for predicated constraints C, written $\kappa \models C$. The rules are given in Figure 4. Rule (V-PCon) requires that for any label solution s_j such that the corresponding predicate P_j "overlaps" with P_i in C (i.e., $\circ\exists(P_i \wedge P_j)$), s_j is a correct solution for the corresponding labeled constraint C_i, written as $[\![C_i]\!]_{s_j}$. Checking a label solution for label constraints ($[\![C]\!]_s$) is more straightforward: we simply substitute constraint variables with their corresponding security levels and check the validity of the result. Here, we use $\langle s, \tau \rangle \Downarrow \ell$ to denote the concrete level of τ under label solution s. For example, $(d > 0 \to \{\alpha_x \mapsto \mathrm{P}, \alpha_y \mapsto \mathrm{S}, \alpha_z \mapsto \mathrm{P}\}, d \le 0 \to \{\alpha_x \mapsto \mathrm{P}, \alpha_y \mapsto \mathrm{P}, \alpha_z \mapsto \mathrm{P}\})$ is a correct solution for the original constraints set (2.1). It is easy to check that the predicates form a partition. Moreover, $d > 0$ intersects with

[1] Another (perhaps more intuitive) definition of solution could be a mapping from each variable to predicated security levels in the form of $\{P_1 \to \ell_1, \ldots, P_n \to \ell_n\}$. It is easy to check that these two forms are interchangeable. We use the form of $\{P_1 \to s_1, \ldots, P_n \to s_n\}$ in this paper to simplify some definitions (e.g., the correctness of a solution).

```
1  // $a, c, d$ : P;  $b$ : S;
2  $x := a + z$;
3  $y := k$; //$k : (d > 0)$?S : P;
4  if ($d > 0$) then $y := b$;
5  if ($d < 0$) then $x := y$;
6  $c := x$;
```

(a) Program

```
1  true $\rightarrow$ P $\sqsubseteq \alpha_x \wedge \alpha_z \sqsubseteq \alpha_x$;
2  $d > 0 \rightarrow$ S $\sqsubseteq \alpha_y$;
3  $\neg(d > 0) \rightarrow$ P $\sqsubseteq \alpha_y$;
4  $d > 0 \rightarrow$ S $\sqsubseteq \alpha_y$;
5  $d < 0 \rightarrow \alpha_y \sqsubseteq \alpha_x$;
6  true $\rightarrow \alpha_x \sqsubseteq$ P;
```

(b) Core Constraints

```
1  P $\sqcup \alpha_z \sqsubseteq \alpha_x$;
2  $(d > 0)$?S : P $\sqsubseteq \alpha_y$;
3  $d > 0 \Rightarrow$ S $\sqsubseteq \alpha_y$;
4  $d < 0 \Rightarrow \alpha_y \sqsubseteq \alpha_x$;
5  $\alpha_x \sqsubseteq$ P;
```

(c) Constraints in Li and Zhang [2017]

```
1  {true} $\cup \alpha_z \leq:_{\{\texttt{true}\}} \alpha_x$;
2  $\{d \leq 0\} \leq:_{\{\texttt{true}\}} \alpha_y$;
3  {false} $\leq:_{\{d>0\}} \alpha_y$;
4  $\alpha_y \leq:_{\{d<0\}} \alpha_x$;
5  $\alpha_x \leq:_{\{\texttt{true}\}}$ {true};
```

(d) Constraints in Murray et al. [2016]

Fig. 5. Program and Constraints using Boolean Expression Predicates.

the predicates of lines 1, 2 in the original constraints; it is easy to check that all label constraints are valid under the label solution in this case. Furthermore, $d \leq 0$ intersects with the predicates of lines 1, 3, 4 in the original constraints; it is easy to check that all label constraints are valid under the label solution in this case as well. Based on Figure 4, we define constraint satisfiability in the standard way:

Definition 2 (Predicated Constraint Satisfiability). *We say a constraint set C is satisfiable, denoted as $\models C$, if and only if exists a correct solution κ:*

$$\models C \iff \exists \kappa.\ \kappa \models C$$

3.3 Expressiveness of the Core Constraint Language

Despite its simplicity, to our best knowledge, the core constraint language can formalize all static information flow analyses with dependent labels. Next, we informally show how to encode four very different kinds of static information flow analyses with dependent labels [Arden et al. 2012; Chen et al. 2018; Ferraiuolo et al. 2017; Li and Zhang 2017; Liu et al. 2009; Lourenço and Caires 2015; Murray et al. 2016; Myers et al. 2006; Polikarpova et al. 2018; Zhang et al. 2015] in the core language. Since the security labels in Lourenço and Caires [2015] are defined as a function from predicates to security levels, a trivial encoding exists. Here, we outline the encoding for other analyses.

Ternary Labels. Following the syntax of ternary expression in C, some prior works [Ferraiuolo et al. 2017; Li and Zhang 2017; Zhang et al. 2015] employ dependent label in form of $b?\tau_1 : \tau_2$, meaning that the concrete security level is τ_1 when b evaluates to true; otherwise the level is τ_2. Consider the program in Figure 5(a), where the security labels of variables a, b, c, d, k are known, while the labels of x, y (α_x and α_y) are to be inferred. The analysis in Li and Zhang [2017] generates the raw constraints (in their syntax) shown in Figure 5(c). For instance, the third constraint $d > 0 \Rightarrow$ S $\sqsubseteq \alpha_y$ is generated from line 4 in the source code: the level of y must be at least as restrictive as that on b whenever the "then" branch is taken. The corresponding constraints in the core language are shown in Figure 5(b), with the following key steps for transformation:

- Lifting dependence: the main mismatch between the raw constraints and our core constraints is that the former allows nested predicates, such as $((d > 0)?$S : P$) \sqsubseteq \alpha_y$. We can lift all

```
// p : l_p;  q : l_q;
test(p)  r = info(p);
else  r = 0;
test(q)  r = r + info(q);
else  r = r + 0;
```

(a) Program

$$\begin{array}{l}(\oplus p,\ \ l_p \le \alpha),\\ (\ominus p,\ \ \mathrm{P} \le \alpha),\\ (\oplus q,\ \ \alpha \sqcup l_q \le \alpha),\\ (\ominus q,\ \ \alpha \sqcup \mathrm{P} \le \alpha),\end{array}$$

(b) Raw Constraints from Chen et al. [2018]

$$\begin{array}{l}p \to l_p \sqsubseteq \alpha;\\ \neg p \to \mathrm{P} \sqsubseteq \alpha;\\ q \to \alpha \sqsubseteq \alpha \wedge l_q \sqsubseteq \alpha;\\ \neg q \to \alpha \sqsubseteq \alpha \wedge \mathrm{P} \sqsubseteq \alpha;\end{array}$$

(c) Core Constraints

Fig. 6. Program and Constraints using Permission Trace.

nested predicates into the predicates in the core language. For example, $((d > 0)?\mathrm{S} : \mathrm{P}) \sqsubseteq \alpha_y$ can be lifted to two constraints: $(d > 0) \to \mathrm{S} \sqsubseteq \alpha_y$ and $\neg(d > 0) \to \mathrm{P} \sqsubseteq \alpha_y$.

- Removing join and meet: when a join operation $\sqcup$ shows up on the left-hand-side of a label constraint, we simply decompose it into two constraints. For example, $L \sqcup \alpha_z \sqsubseteq \alpha_x$ decomposes into $L \sqsubseteq \alpha_x \wedge \alpha_z \sqsubseteq \alpha_x$. The dual can be done for the meet operation on the right-hand-side.
- Default predicate: without any predicate, a raw constraint means that the label constraint holds under all conditions. We make this explicit by adding a predicate `true` for such constraints.

Predicate as Labels. When a security lattice only has two labels P and S, some prior works use predicates to specify dependent labels [Murray et al. 2016; Polikarpova et al. 2018]. A predicate P in those analyses can be interpreted as P when P holds; otherwise, the level is S. For example, $\{d \le 0\}$ represents a dependent label that when $d \le 0$, the level is P; otherwise, the level is S. The raw constraints from Murray et al. [2016] are shown in Figure 5(d), where constraints are in form of $P_1 \le:_P P_2$, where the P serves as the condition under which the restriction $P_1 \le: P_2$ holds. Their analysis uses $\cup$ operation when multiple variables are used in an expression, such as line 2 in the example. To encode such constraints, the key steps are:

- Union decomposition: unions semantically means "join" on security labels. Hence, we can decompose them in a way similar to encoding join. For example, $\{\mathtt{true}\} \cup \alpha_z \le:_{\{\mathtt{true}\}} \alpha_x$ decomposes into $\{\mathtt{true}\} \le:_{\{\mathtt{true}\}} \alpha_x \wedge \alpha_z \le:_{\{\mathtt{true}\}} \alpha_x$.
- Lifting predicates: similar to lifting predicates for ternary labels, both P_1 and P_2 can be lifted into P. For example, $\{d \le 0\} \le:_{\{\mathtt{true}\}} \alpha_y$ is encoded to $\{\mathtt{true}\} \le:_{\{d \le 0\}} \alpha_y$ and $\{\mathtt{false}\} \le:_{\{\neg(d \le 0)\}} \alpha_y$.
- Replacing predicates with levels: after the previous two steps, other than constraint variables, the remaining label constraints only involve $\{\mathtt{true}\}$ or $\{\mathtt{false}\}$, where we replace the former with P and the latter with S. For instance, $\{\mathtt{true}\} \le:_{\{d \le 0\}} \alpha_y$ is encoded as $d \le 0 \to \mathrm{P} \sqsubseteq \alpha_y$.

The encoded constraints in the core language are shown in Figure 5(b). This is identical to the encoding of the raw constraints in Figure 5(c), suggesting a strong connection between the ternary labels and predicates as labels. Though the analysis in Polikarpova et al. [2018] differs in certain ways, its constraints can be encoded in a similar way.

Permission Predicates. Security level in Chen et al. [2018] depends on permissions granted to a program. Their analysis generates constraints in form of $(P,\ \tau_1 \le \tau_2)$, corresponding to $P \to \tau_1 \sqsubseteq \tau_2$ in the core constraint language except that P is a set of granted permissions (written as $\oplus p$) or permissions that are not granted (written as $\ominus p$). Consider the program in Figure 6(a), adapted from Chen et al. [2018]. The security label of p, q are known as l_p, l_q and the label of r is unknown and annotated as α. The encoding (shown in Figure 6(c)) is straightforward, where permissions are encoded as boolean expressions in the core language.

Dynamic Labels. Dynamic labels that can be found in Jif [Myers et al. 2006] and its extensions [Arden et al. 2012; Liu et al. 2009] has a restricted form of dependence: it depends on a label-typed variable. For example, a constraint $\alpha_x \sqsubseteq l$ where l is a label-typed program variable specifies a constraint that the level of x must be at most as restrictive as the run-time value of l when the corresponding program statement is executed. For any finite lattice, such constraints can be encoded by enumerating all possible values of l: $\alpha_x \sqsubseteq l$ can be encoded as $l = \mathrm{P} \rightarrow \alpha_x \sqsubseteq \mathrm{P}$ and $l = \mathrm{S} \rightarrow \alpha_x \sqsubseteq \mathrm{S}$ for a two-level lattice $\{\mathrm{P}, \mathrm{S}\}$. We note that such a naive encoding might be inefficient for a complex lattice, since the encoding requires one extra constraint for each level in the lattice. One more efficient alternative is to enrich the label constraints to include extra assumptions on label-typed variables and utilize more sophisticated label-constraint solvers (e.g. the solver in Jif and the SHErrLoc solver [Zhang and Myers 2014]) for the enriched label constraints with assumptions. However, since the efficiency of encoding is largely an orthogonal issue, we leave that as future work.

Converting Solution Back. In most cases of information flow analysis, what matters is the existence of a solution (meaning that the program being verified is secure), rather than what a solution it is. However, we note that if needed, a solution in the core language can be decoded as well, mostly following the same idea for encoding. Taking the analysis in Li and Zhang [2017] as an example. For any constraint variable α, a solution $\kappa = \{P_1 \rightarrow s_1, \ldots, P_n \rightarrow s_n\}$ can be decoded as a ternary label $P_1 ? s_1(\alpha) : \ldots P_{n-1} ? s_{n-1}(\alpha) : s_n(\alpha)$ (recall that $\mathbb{P}_\kappa$ is a partition).

3.4 Alternative Formalizations

Various forms of constraints have been used for program verification tasks. However, to the best of our knowledge, no existing form of constraints can directly handle dependent security labels with an arbitrary security lattice. Next, we discuss the major hurdles of handling dependent security labels via existing forms of constraints.

SMT Constraint. It is possible to encode dependent label subtyping as an SMT constraint with alternating quantifiers; however, such constraints are in general undecidable and handled quite poorly in practice. This stands in contrast to label subtyping without dependency, which can be encoded using efficiently-solvable, quantifier-free SMT constraints. For example, a (predicated) constraint $(b = 1 \rightarrow \alpha \sqsubseteq \mathrm{P}) \wedge (b \neq 1 \rightarrow \mathrm{S} \sqsubseteq \alpha)$ has a dependent solution that α is P whenever $b = 1$; α is S whenever $b \neq 1$. But an SMT solver with theory on semi-lattice will simply reject the constraint, since neither P nor S (without dependence) is a solution for α.

Constraint Horn Clauses (CHCs). CHCs is an intermediate constraint format commonly used in functional verification. We note that in the simplest case of two-level security lattice, CHCs is a promising alternative formalism of dependent security labels. For instance, dependent labels can be encoded by predicates, as shown in Figure 5(d), which is adapted from prior work [Murray et al. 2016]. Recall that the satisfaction of a predicate encodes P in such encoding. Hence, with form of $P_1 \leq:_P P_2$, the constraints can be converted to $P \wedge P_2 \Rightarrow P_1$ in CHCs. Lifty [Polikarpova et al. 2018] employs a similar idea by encoding two security levels P and S into tagged types, with predicates over stores and users. Lifty employs an inference engine that transforms constraints into Horn Clauses. Although CHCs is a neat formalism for a two-level lattice, extending it to encode constraints with a general security lattice is still an open and challenging problem.

4 THE DERIVATION FRAMEWORK

The derivation framework enables the design and validation of label inference algorithms that transform original constraints into a simplified form where a solution is more feasible to obtain. In this section, we first describe three approaches under this model and then develop a proof framework that validates the soundness and completeness of algorithms under the derivation framework.

4.1 Derivation Framework and Its Instances

We first explore the large space of inference algorithms under our derivation framework by giving derivation examples and show why they are useful. We defer the discussion of how to check their soundness and completeness to Section 4.2.

We first define *derivations* as constraint set transformation:

Definition 3 (Derivation). *A derivation abstracts a transformation from one constraint set $C1$ to another constraint set $C2$, denoted as $C1 \rightrightarrows C2$.*

Sound Derivations and Early-Accept Approach. For a derivation $C1 \rightrightarrows C2$, we say it is *sound* if any solution of $C2$ is a solution of $C1$ (intuitively, the derived constraints are stronger). Sound derivations are useful since if a constraint solving algorithm derives constraints in a sound way, the algorithm may terminate early, as long as there is a solution on the current derivation. We refer to such an approach as the *early-accept* approach.

The early-accept approach is an iterative approach where a sound derivation is employed at each iteration. To make the algorithm complete eventually, at each iteration, an early-accept algorithm weakens the constraints in current iteration to produce weaker yet sound derivation for the next iteration. If the final iteration is a sound and complete derivation, the algorithm becomes both sound and complete.

A set of sound derivations are shown in Figure 1. At iteration 1, the derived constraint set simply removes all predicates and check if the constraints can be solved without any dependence. This derivation is unsatisfiable, thus, a weaker derivation is employed in iteration 2. At iteration 2, constraints are solved under predicate $d > 0$ and its negation. This sound derivation has a solution (the solution shown in Section 3.2 already). Hence, the algorithm stops with the solution. If the derivation in iteration 2 were unsatisfiable, the algorithm continues to derivation 3, which is both sound and complete.

When a set of constraints can be solved with a small number of dependences, the early-accept approach might find a solution early. However, in the worst case (e.g., when constraints are unsatisfiable), a sound and complete algorithm may either find a solution or find there is no solution in the last derivation, which wastes computation resources.

Complete Derivation and Early-Reject Approach. For a derivation $C1 \rightrightarrows C2$, we say the derivation is *complete* if when there is no solution of $C2$, then $C1$ must be unsatisfiable (intuitively, the derived constraints are weaker). Complete derivations are useful since if a constraint solving algorithm derives constraints in a complete way, the algorithm may terminate early, as long as there is no solution on the current derivation. We refer to such an approach as the *early-reject* approach.

The early-reject approach is dual to the early-accept approach. Starting from a complete derivation, an early-reject algorithm works on complete derivations in each iteration. If the final iteration has a sound and complete derivation, then the algorithm is both sound and complete. For example, an unsatisfiable constraint set and its complete derivations are shown in Figure 7.

Unsatisfiable Constraint Set

$$a \leq 0 \rightarrow \mathrm{S} \sqsubseteq \alpha \land \alpha \sqsubseteq \mathrm{S}$$
$$a \geq 0 \rightarrow \alpha \sqsubseteq \mathrm{P} \tag{4.1}$$

Derivaton 1 (A complete derivation)

$$a \leq 0 \rightarrow \mathrm{S} \sqsubseteq \alpha \tag{4.2}$$

Derivation 2: (A stronger yet complete derivation)

$$a < 0 \rightarrow \mathrm{S} \sqsubseteq \alpha \land \alpha \sqsubseteq \mathrm{S}$$
$$a = 0 \rightarrow \mathrm{S} \sqsubseteq \alpha \land \alpha \sqsubseteq \mathrm{S} \tag{4.3}$$
$$a > 0 \rightarrow \alpha \sqsubseteq \mathrm{P}$$

Final Derivation:(A sound and complete derivation)

$$a < 0 \rightarrow \mathrm{S} \sqsubseteq \alpha \land \alpha \sqsubseteq \mathrm{S}$$
$$a = 0 \rightarrow \mathrm{S} \sqsubseteq \alpha \land \alpha \sqsubseteq \mathrm{S} \land \alpha \sqsubseteq \mathrm{P} \tag{4.4}$$
$$a > 0 \rightarrow \alpha \sqsubseteq \mathrm{P}$$

Fig. 7. Constraints and Complete Derivations.

Figure 7, derivations 1 and 2 are satisfiable. Hence, an early-reject algorithm continues to the final derivation, where no solution exists due to the second constraint $a = 0 \rightarrow \mathrm{S} \sqsubseteq \alpha \land \alpha \sqsubseteq \mathrm{S} \land \alpha \sqsubseteq \mathrm{P}$. Hence, the algorithm rejects the original constraints.

Compared with early-accept, early-reject has the potential to reject a constraint set early. However, in the worst case (e.g., when constraints are satisfiable), a sound and complete algorithm may waste considerable computation resources in the complete but not sound derivations.

One-Shot Approach. To achieve soundness and completeness, one direction is to start from a sound (resp. complete) derivation, and eventually reach a sound and complete derivation, as sketched above. Another direction is to transform the original constraints directly into an equivalent constraint set. We call such an approach the *one-shot* approach.

The *one-shot* approach uses one equivalent derivation in constraint solving. Consider the constraints in 5(b). The algorithm by Chen et al. [2018] (an instance of one-shot approach) enumerates all combinations of predicates and their negations, generating an equivalent derivation:

$$\{\, p \land q \rightarrow l_p \sqsubseteq \alpha \land l_q \sqsubseteq \alpha; \quad p \land \neg q \rightarrow l_p \sqsubseteq \alpha; \quad \neg p \land q \rightarrow l_q \sqsubseteq \alpha; \quad \neg p \land \neg q \rightarrow \mathrm{P} \sqsubseteq \alpha; \,\}$$

These constraints can be solved with a solution that combines local solutions under each predicate.

Both the advantage and the disadvantage of this approach are conspicuous: the transformations are simple and intuitive to implement; however, the number of transformed constraints grows exponentially: given n different predicates, the transformation results in $O(2^n)$ constraints under different predicates; this is confirmed in our evaluation (Section 6).

4.2 Proof Framework

The derivation framework allows a large space for constraint solving algorithms. One crucial question for those potential algorithms is that whether a derivation is *sound* (i.e., any solution of a derivation is a solution of the previous derivation) and/or *complete* (i.e., no solution of a derivation implies no solution of the previous derivation). We identify a few key properties to make it easy to check if a derivation is sound and/or complete.

Soundness. A derivation is *sound* if the derived constraints are stronger:

Definition 4 (Sound Derivation). *We say a derivation is sound, if any solution of the derived constraint set is also a solution of the original constraint set:*

$$\forall C1 \rightrightarrows C2, \kappa.\ \kappa \models C2 \Rightarrow \kappa \models C1$$

To make a derivation sound, we first note that the derived set should at least "cover" the same or more predicates in the original set, as illustrated to the left of Figure 8. The reason is that a solution

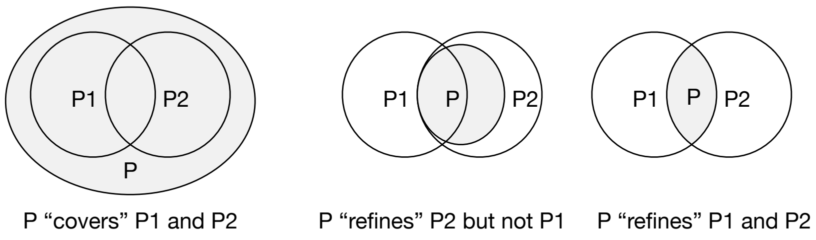

Fig. 8. Illustrated Restrictions on Predicates.

on the stronger set should consider all cases constrained in the original set. For example, consider the (unsound) derivation, (4.1) $\rightrightarrows$ (4.2) in Figure 7, where the case $a > 0$ is not covered in (4.2). Then, a solution of (4.2) is not necessarily a solution of (4.1), since the former is "less constrained" under the uncovered case $a > 0$. This requirement is formalized as the *Coverage* property on the predicate set:

Property 1 (Coverage). *We say a predicate set $\mathbb{P}_2$ covers a predicate set $\mathbb{P}_1$, denoted as $\mathbb{P}_2 \gtrdot \mathbb{P}_1$, if they satisfy:*

$$\circ\forall(\bigvee \mathbb{P}_1 \Rightarrow \bigvee \mathbb{P}_2)$$

Moreover, for soundness, each label constraint in the original set should be "projected" (i.e., propagated) to the derived set. That means for each predicated constraint $P_i \rightarrow C_i$ in the derived set, C_i should be stronger than any C_j in the original set if $P_j \rightarrow C_j$ is in the original set and P_i, P_j may occur at the same time (i.e., $\circ\exists(P_i \wedge P_j)$). Consider the unsound derivation (4.1) $\rightrightarrows$ (4.3) and the sound derivation (4.1) $\rightrightarrows$ (4.4). We notice that the constraint $\alpha \sqsubseteq$ P under $a \geq 0$ is not included under $a = 0$ in the (unsound) constraints (4.3). To check that the derivation (4.1) $\rightrightarrows$ (4.4) is sound, we note that $a = 0$ has intersection with two predicates $a \leq 0$ and $a \geq 0$ in (4.1). Moreover, the label constraints under $a = 0$ involve all label constraints in (4.1) (i.e., the derived label constraints are stronger). Hence, the derivation (4.1) $\rightrightarrows$ (4.4) is sound. We formalize this observation as the *Projection* property:

Property 2 (Projection). *We say a constraint set C is projected to a constraint $P \rightarrow C$, denoted as $C \rightsquigarrow^{\dagger} P \rightarrow C$, if they satisfy:*

$$\forall s.\ [\![C]\!]_s \Rightarrow [\![\bigwedge_{P_i \rightarrow C_i \in C \wedge \circ\exists(P \wedge P_i)} C_i]\!]_s$$

Note that when P has no intersection with any predicate in $\mathbb{P}_C$, the requirement is $\forall s.\ [\![C]\!]_s \Rightarrow [\![\emptyset]\!]_s$. Since any solution works on $\emptyset$, any C suffice in this case. Intuitively, this does not break soundness since the original constraints put no restriction under P anyway. We can lift the projection definition to constraint sets:

Property 3 (Full Projection). *We say constraint set $C1$ is fully projected to $C2$, denoted as $C1 \rightsquigarrow^{\dagger} C2$, if every constraint in $C1$ is projected to constraints in $C2$:*

$$\forall P_2 \rightarrow C_2 \in C2.\ C1 \rightsquigarrow^{\dagger} P_2 \rightarrow C_2$$

Altogether, a derived constraint is sound if (1) the new predicates *cover* the original predicates, and (2) the original constraint set is *fully projected* to the derived constraint set:

Theorem 1 (Soundness). *If the original constraint set is covered by and fully projected to the derived constraint set, then the derivation is sound:*

$$\forall C1 \rightrightarrows C2, \kappa.\ C1 \rightsquigarrow^{\dagger} C2 \wedge \mathbb{P}_{C2} \gtrdot \mathbb{P}_{C1} \wedge \kappa \models C2 \Rightarrow \kappa \models C1$$

Proof. By the definition of $\kappa \models C1$, we need to show that $\forall P_i \rightarrow C_i \in C1, P_k \rightarrow s_k \in \kappa.\ \circ\exists(P_i \wedge P_k) \Rightarrow [\![C_i]\!]_{s_k}$. That is, we need to show $[\![C_i]\!]_{s_k}$ under the condition that $\circ\exists(P_i \wedge P_k)$. By Property 1, we know that $\circ\forall(\bigvee \mathbb{P}_{C1} \Rightarrow \bigvee \mathbb{P}_{C2})$. Hence, $\circ\forall(P_i \wedge P_k \Rightarrow P_i \Rightarrow \bigvee \mathbb{P}_{C1} \Rightarrow \bigvee \mathbb{P}_{C2})$. Therefore, there there must be some $P_j \in \mathbb{P}_{C2}$ such that $\circ\exists(P_k \wedge P_j \wedge P_i)$. That is, we have $\circ\exists(P_k \wedge P_j)$ and $\circ\exists(P_j \wedge P_i)$. From the former, we have $[\![C_j]\!]_{s_k}$ due to validation rule V-PCon and the fact $\kappa \models C2$; from the latter and Property 2 and 3, we have $[\![C_j]\!]_{s_k} \Rightarrow [\![C_i]\!]_{s_k}$. Therefore, we proved that $[\![C_i]\!]_{s_k}$. □

Completeness. A derivation is *complete* if its derived constraint set is weaker:

Definition 5 (Complete Derivation). *We say a derivation is complete, if any solution of the original set is also a solution of the derived set:*

$$\forall C1 \rightrightarrows C2, \kappa.\ \kappa \models C1 \Rightarrow \kappa \models C2$$

To make a derivation complete, there is one restriction on the predicate sets of $C2$ (i.e., there is no dual to the coverage property). The reason is that, a local view is sufficient in rejecting a constraint set. For example,

$$a = 0 \rightarrow \mathrm{S} \sqsubseteq \alpha \wedge \alpha \sqsubseteq \mathrm{S} \wedge \alpha \sqsubseteq \mathrm{P}$$

which is unsatisfiable, allows us to reject (4.1) since when $a = 0$, all label constraints on the right are required in (4.1), but they are not satisfiable.

To make a derivation complete, each label constraint with precondition P should be *weaker* than the intersection of all C_i where P "refines" the corresponding P_i (i.e., P_i "includes" P, or $\circ\forall(P \Rightarrow P_i)$), as illustrated in the middle of Figure 8. For example, (4.2) is a complete derivation of (4.1) according to this observation: first, $a \leq 0$ refines $a \leq 0$ but not $a \geq 0$ in (4.1); second, $\mathrm{S} \sqsubseteq \alpha \wedge \alpha \sqsubseteq \mathrm{S} \Rightarrow \mathrm{S} \sqsubseteq \alpha$ (i.e., $\mathrm{S} \sqsubseteq \alpha$ is weaker than the corresponding label constraint of $a \leq 0$ in (4.1)). However, a similar derivation $\{a \leq 0 \Rightarrow \alpha \sqsubseteq \mathrm{P}\}$ is not a complete derivation from (4.1), since its label constraint $\alpha \sqsubseteq \mathrm{P}$ is not weaker than the original one under the same precondition. This observation is formalized as the *Inferred* property:

Property 4 (Inferred Constraint). *We say a constraint $P \rightarrow C$ is inferred from a constraint set C, denoted as $C \rightsquigarrow^{\diamond} P \rightarrow C$, if they satisfy:*

$$\forall s.[\![\bigwedge_{P_i \rightarrow C_i \in C \wedge \circ\forall(P \Rightarrow P_i)} C_i]\!]_s \Rightarrow [\![C]\!]_s$$

This property is a dual of the projection property 2. When P does not refine any predicate in $\mathbb{P}_C$, the requirement is $\forall s.\ [\![\emptyset]\!]_s \Rightarrow [\![C]\!]_s$. Since any solution works on $\emptyset$, this property requires $\forall s.\ [\![C]\!]_s$, which essentially makes it impossible to reject the original constraints due to $P \rightarrow C$. We can lift the projection definition to constraint sets:

Property 5 (Fully-Inferred). *We say constraint set $C2$ is fully-inferred from constraint set $C1$, denoted as $C1 \rightsquigarrow^{\diamond} C2$ if all constraints are inferred constraints:*

$$\forall P_i \rightarrow C_i \in C2.\ C1 \rightsquigarrow^{\diamond} P_i \rightarrow C_i$$

A derivation is complete if it is *fully-inferred*:

Theorem 2 (Completeness). *If the derived constraint set is fully-inferred from the original constraint set, then the derivation is complete:*

$$\forall C1 \rightrightarrows C2, \kappa.\ C1 \rightsquigarrow^{\diamond} C2 \wedge\ \kappa \models C1 \Rightarrow \kappa \models C2$$

Proof. By the definition of $\kappa \models C2$, we need to show that $\forall P_j \to C_j \in C2, P_k \to s_k \in \kappa.\ \circ\exists(P_j \wedge P_k) \Rightarrow [\![C_j]\!]_{s_k}$. Consider the set $Q = \left\{P_i \mid P_i \in \mathbb{P}_{C1} \wedge \circ\forall(P_j \Rightarrow P_i)\right\}$. When $Q = \emptyset$, we have $C_j = \emptyset$ by Property 4. Hence, $[\![C_j]\!]_{s_k}$ is vacuously true.

Otherwise, for any $P_i \in Q$, we have $\circ\forall(P_j \Rightarrow P_i)$. Given $\circ\forall(P_j \Rightarrow P_i)$ and $\circ\exists(P_j \wedge P_k)$, it must be true that $\circ\exists(P_j \wedge P_i \wedge P_k)$, which implies $\circ\exists(P_i \wedge P_k)$. Since $\kappa \models C1$, we have $[\![C_i]\!]_{s_k}$ by validation rule V-PCon. Since this is true for any $C_i \in Q$, we have $\bigwedge_{C_i \in Q} [\![C_i]\!]_k$. From Properties 4 and 5, we know that $\forall s.\ [\![\bigwedge_{C_i \in Q} C_i]\!]_s \Rightarrow [\![C_j]\!]_s$. Hence, $[\![\bigwedge_{C_i \in Q} C_i]\!]_{s_k} \Rightarrow [\![C_j]\!]_{s_k}$, and therefore, $[\![C_j]\!]_{s_k}$. □

Equivalence. A derivation is equivalent if it is both sound and complete. We can check so by all of the properties defined above. However, in practice, it is more desirable to develop algorithms that start from sound (or complete) only derivations, and evolve to a sound and complete derivation eventually, with respect to the original constraint set (as illustrated in Figure 2).

To make it happen, we first define the *weakest* derivation that complies Property 3 and the *strongest* derivation that complies Property 5 :

Definition 6 (Weakest Sound Derivation). *A derivation is the weakest sound derivation, denoted as $C1 \rightrightarrows^{\dagger} C2$, if*

$$\mathbb{P}_{C2} \gtrdot \mathbb{P}_{C1} \text{ and } \forall P_2 \to C_2 \in C2.\ C_2 = \bigwedge_{P_i \to C_i \in C1 \wedge \circ\exists(P_i \wedge P_2)} C_i$$

By construction, the soundness of this derivation is trivial. Similarly, we define the strongest complete derivation, whose completeness directly follows its construction:

Definition 7 (Strongest Complete Derivation). *A derivation is the strongest complete derivation, denoted as $C1 \rightrightarrows^{\diamond} C2$, if*

$$\forall P_2 \to C_2 \in C2.\ C_2 = \bigwedge_{P_i \to C_i \in C1 \wedge \circ\forall(P_2 \Rightarrow P_i)} C_i$$

What makes weakest sound derivation and strongest complete derivation interesting is that they are interchangeable when for any $P_1 \in \mathbb{P}_{C1}$ and $P_2 \in \mathbb{P}_{C2}$, P_2 refines P_1 whenever they intersect, as illustrated to the right of Figure 8. This is called the *Refinement* property:

Property 6 (Refinement). *We say a predicate set $\mathbb{P}_2$ is a refinement of a predicate set $\mathbb{P}_1$, denoted as $\mathbb{P}_1 \ll \mathbb{P}_2$, if we have*

(1) $\forall P_2 \in \mathbb{P}_2, \circ\exists(P_2)$
(2) $\forall P_1 \in \mathbb{P}_1, P_2 \in \mathbb{P}_2.\ \circ\exists(P_1 \wedge P_2) \implies \circ\forall(P_2 \Rightarrow P_1)$

Theorem 3. *A refined weakest sound derivation is a strongest complete derivation:*

$$\forall C1 \rightrightarrows^{\dagger} C2, \kappa.\ \mathbb{P}_{C1} \ll \mathbb{P}_{C2} \Rightarrow C1 \rightrightarrows^{\diamond} C2$$

Proof. Consider any $P_2 \in \mathbb{P}_{C2}$. Consider any $P_1 \in \mathbb{P}_{C1}$ such that $\circ\forall(P_2 \Rightarrow P_1)$. Since $\circ\exists(P_2)$, we know that $\circ\exists(P_2 \wedge (P_2 \Rightarrow P_1)) = \circ\exists(P_2 \wedge P_1)$. Hence, the refinement property implies that $\circ\exists((P_1 \wedge P_2)) \iff \circ\forall(P_2 \Rightarrow P_1)$. Hence, definitions 6 and 7 coincide.

□

This theorem provides a constructional strategy for evolving sound derivations to a sound and complete derivation. For example, given original predicates $\mathbb{P}_C = \{P_1, P_2\}$, the following sequence of covers of $\mathbb{P}_C$ eventually satisfies the refinement property: $\{\texttt{true}\}$, $\{P_1, \neg P_1\}$, $\{P_1 \wedge P_2, P_1 \wedge \neg P_2, \neg P_1 \wedge P_2, \neg P_1 \wedge \neg P_2\}$. We discuss further on such an inference algorithm in Section 5.2.

On the other hand, a complete derivation becomes sound and complete if it satisfies refinement and covers the original predicate set:

Theorem 4. *A refined strongest complete-derivate is a weakest sound-derivation, if the derived constraint set covers the original one*

$$\forall C1 \rightrightarrows^{\diamond} C2, \kappa.\, \mathbb{P}_{C1} \ll \mathbb{P}_{C2} \land \mathbb{P}_{C2} \succ \mathbb{P}_{C1} \;\Rightarrow C1 \rightrightarrows^{\dagger} C2$$

Proof. The proof is similar to that of Theorem 3. Note that the coverage requirement in weakest sound derivation is in the assumption. □

5 ALGORITHMS

There are many approaches to utilize the derivation framework. In this section, we discuss how to instantiate the major components of the framework, as well as provide four concrete constraint solving algorithms.

5.1 Partition Method

The general framework allows great flexibility in the predicate set $\mathbb{P}$ of a derivation. Hereafter, we explore one approach, where the predicate set $\mathbb{P}$ forms a partition (Definition 1) for two reasons: (1) a partition satisfies the coverage requirement for sound derivation as well as sound and complete derivations, and (2) a partition allows a solver to solve label constraints under each predicate in isolation and combine sub-solutions to one solution. We call the procedure of generating predicate set $\mathbb{P}$ of a derivation as a partition *partition method.*

Intuitively, a partition method controls the pace of the solving process (as illustrated by the lines in Figure 2): in the one-shot approach, partition method directly generates a refinement of the original constraint set; in early-accept or early-reject approach, partition algorithm is responsible to provide a different set of predicates at each iteration and also guarantee that the final predicate set is a refinement of the original constraint set.

Since the one-shot approach is straightforward, we next focus on two concrete partition algorithms, following two different intuitions:

- *Sequential Partitioning* is an avid algorithm devoted to reaching the final refinement with the least efforts wasted on the path;
- *Combinational Partitioning* is more adventurous, trying out all kindred cases along the way.

Both algorithms start from singleton partition {`true`} and eventually reach the same refinement of the original constraint set at the final iteration. We know when one predicate P_1 is given, the space can be partitioned into two parts, P_1 and $\neg P_1$. When two predicates P_1, P_2 are given, the space can be at most partitioned into four parts: $P_1 \land P_2$, $P_1 \land \neg P_2$, $\neg P_1 \land P_2$ and $\neg P_1 \land \neg P_2$. We call this relationship *Partition Space*:

Definition 8 (Partition Space). *Given predicates $P_1, P_2, ..., P_n$, a partition space of these predicates, denoted as $\boxplus(P_1, P_2, ..., P_n)$ is a set of predicates after partitioned :*

$$\boxplus\,(P_1, P_2, ..., P_n) = \{\, p_1 \land p_2 \land \ldots \land p_n \mid p_1 \in \{P_1, \neg P_1\}, p_2 \in \{P_2, \neg P_2\}, ..., p_n \in \{P_n, \neg P_n\}\}$$

Given predicates $P_1, \ldots, P_n$ in the original set, the sequential partitioning algorithm constructs predicate sets {`true`}, $\{P_1, \neg P_1\}$, $\{P_1 \land P_2, P_1 \land \neg P_2, \neg P_1 \land P_2, \neg P_1 \land \neg P_2\}, \ldots,$ while the combinational partitioning algorithm constructs sets {`true`}, $\{P_1, \neg P_1\}$, $\{P_2, \neg P_2\}, \ldots$ as follows.

Sequential Partitioning. Sequential partitioning refines the partition sequentially one predicate at a time. For a predicate set $\{P_1, P_2, ..., P_n\}$, the sequential partitioning algorithm produces:

Iteration 0: Starting : {`true`}
Iteration 1: Partition by P_1 : $\boxplus(P_1)$
Iteration 2: Partition by P_1, P_2: $\boxplus(P_1, P_2)$

...

Iteration n: Partition by $P_1, P_2, \ldots, P_n$: $\boxplus(P_1, P_2, \ldots P_n)$

Consider the predicates used in the sound derivations in Figure 1. It follows a sequential partitioning: derivation (2.2) uses $\{\texttt{true}\}$, derivation (2.3) uses $\{d > 0, d \leq 0\}$, and derivation (2.4) uses $\{d > 0 \land d < 0, d > 0 \land d \geq 0, d \leq 0 \land d < 0, d \leq 0 \land d \geq 0\}$, which is equivalent to $\{d > 0, d = 0, d = 0\}$. The last predicate set is a refinement of the original predicates.

While sequential partitioning is intuitive and simple to implement, we observe that it is not stable at performance: the execution time of constraint solving may differ dramatically depending on the order of the input predicates. Consider a special case that only the sound derivation partitioned by P_1 can be solved. In the best case, sequential partitioning picks P_1 in the first iteration, resulting a partition $\boxplus(P_1)$ of size 2; however, in the worse case, P_1 gets picked at the last iteration. In this case, the partition algorithm will go generate a last-level partition $\boxplus(P_1, P_2, \ldots, P_n)$, with 2^n predicated constraints.

Combinational Partitioning. Combinational partitioning is designed to be more stable for the performance. It tries out all partitions at the same level before going to the next level. For a predicates set: $\{P_1 \to C_1, P_2 \to C_2, \ldots, P_n \to C_n\}$, the partition procedure is as follows:

Iteration 0: Starting : $\{\texttt{true}\}$
Comb 1 - Iteration 1: Partition by P_1: $\boxplus(P_1)$
Comb 1 - Iteration 2: Partition by P_2: $\boxplus(P_2)$
...
Comb 1 - Iteration n: Partition by P_n: $\boxplus(P_n)$
Comb 2 - Iteration 1: Partition by P_1, P_2 : $\boxplus(P_1, P_2)$
Comb 2 - Iteration 2: Partition by P_1, P_3 : $\boxplus(P_1, P_3)$
...
Comb 2 - Iteration C_n^2: Partition by P_{n-1}, P_n : $\boxplus(P_{n-1}, P_n)$
...
Comb n - Iteration 1: Partition by $P_1, P_2, \ldots, P_n$: $\boxplus(P_1, P_2, \ldots P_n)$

For the special case that only the sound derivation partitioned by P_1 can be solved, in the worse case, the combinational partition picks P_1 at **Comb 1 - Iteration n** with partition $\boxplus(P_1)$ of size 2. Though the sequential partitioning also finds the result at the n-th iteration, the size of the partition set grows exponentially at each iteration. For combinational partitioning, partition in **Comb 1** are all of size 2; thus, the efforts to reach the n-th iteration is just linear, rather than exponential.

5.2 Derivation Method

A derivation method, the core component of a solving algorithm, constructs derivations to be checked under the current iteration. In particular, a derivation method takes in two parameters: the original predicated constraint set $\{P_1 \to C_1, \ldots, P_n \to C_n\}$, as well as a predicate set $\mathbb{P}$ (constructed by a partition method from Section 5.1), under which a set of label constraints are to be generated.

For each predicate $P \in \mathbb{P}$, a sound derivation method (SoundDerive) computes predicated constraints under P following Definition 6:

$$\textsc{SoundDerive}(\{P_1 \to C_1, \ldots, P_n \to C_n\}, \mathbb{P}) \triangleq \bigcup_{P \in \mathbb{P}} \{P \to C_i \mid \circ\exists(P \land P_i)\}$$

Similarly, a complete derivation method (CompleteDerive) computes predicated constraints under P following Definition 7:

$$\textsc{CompleteDerive}(\{P_1 \to C_1, \ldots, P_n \to C_n\}, \mathbb{P}) \triangleq \bigcup_{P \in \mathbb{P}} \{P \to C_i \mid \circ\forall(P \Rightarrow P_i)\}$$

Since we simply construct the sound/complete sets as the weakest sound derivation/strongest complete derivation (Definitions 6 and 7), their correctness is easy to validate. Since the partition method ensures coverage and refinement in the last step, both derivations are eventually sound and complete (Theorems 3 and 4).

5.3 Constraint Solving Algorithms

Based on derivation and partition methods, we present four concrete algorithms.

- *One-Shot Algorithm*: this algorithm is equivalent to the solver in Chen et al. [2018]. It directly works on a refinement of the original constraint set.
- *Iterative Early-Accept Algorithm*: this algorithm iteratively uses weakest sound derivations. Hence, if an iteration is satisfiable, the algorithm accepts the original constraint set; otherwise, the algorithm continues until the final equivalent derivation.
- *Iterative Early-Reject Algorithm*: this algorithm iteratively uses strongest complete derivations. Hence, if an iteration is unsatisfiable, the algorithm rejects the original constraint set; otherwise, the algorithm continues until the final equivalent derivation.
- *Hybrid Algorithm* The hybrid algorithm takes advantages of both sound and complete derivations and seeks for early termination whenever possible.

Next, we discuss each algorithm in details.

One-Shot Algorithm. One-shot algorithm directly works on a refinement of the original constraint set, where its sound derivation is equivalent to its complete derivation. The pseudo code is shown in Algorithm 1. It first initializes the partition algorithm and obtains the final refinement partition

Algorithm 1 One-Shot Solver

```
1: function SOLVE-ONE-SHOT($cons\text{-}set$)
2:     $solution \leftarrow \emptyset$
3:     $partition$.INIT($cons\text{-}set$)
4:     $partt \leftarrow partition$.FINAL-PARTT()
5:     $derive \leftarrow$ SOUNDDERIVE($cons\text{-}set, partt$)
6:     for $i \leftarrow 0, length(partt)$ do
7:         if $rmSolver$.SOLVE($derive[i]$) then
8:             $solution[partt[i]] \leftarrow rmSolver$.SOLUTION()
9:         else
10:            $unsolve \leftarrow partt[i]$
11:            return false
12:        end if
13:    end for
14:    return true
15: end function
```

(Section 5.1). Sound derivation method is then employed at line 5 to generate label constraints for each predicate in the partition set.

At line 7, a label constraint solver, *rmSolver* (an implementation of the Rehof-Mogensen algorithm [Rehof et al. 1999]) is used to solve label constraints under each predicate. *rmSolver* solves label constraints without predicates. Since the derivation at line 5 is indeed equivalent, if any constraint is unsatisfiable, the original set is rejected at line 11. Only when all label constraints are solved, a global solution is found by combining the local solutions.

Iterative Early-Accept Algorithm. Early-accept algorithm makes use of sound derivations so that if any sound derivation is satisfiable, the algorithm accepts the constraint set with the solution found; otherwise, the algorithm continues until the final equivalent derivation. The pseudo code is show in Algorithm 2.

Algorithm 2 Early-Accept Solver

```
1:  function SOLVE-EARLY-ACCEPT($cons\text{-}set$)
2:      $new\text{-}partt, solution \leftarrow \emptyset$
3:      $partition$.INIT($cons\text{-}set$)
4:      while $partition$.STOP() == false do
5:          $partt \leftarrow partition$.NEXT-PARTT($new\text{-}partt$)
6:          $new\text{-}partt \leftarrow \emptyset$
7:          $derive \leftarrow$ SOUNDDERIVE($cons\text{-}set, partt$)
8:          for $i \leftarrow 0, length(partt)$ do
9:              if $rmSolver$.SOLVE($derive[i]$) then
10:                 $solution[partt[i]] \leftarrow rmSolver$.SOLUTION()
11:             else
12:                 $new\text{-}partt$.APPEND(partt[i])
13:             end if
14:         end for
15:         if $new\text{-}partt == \emptyset$ then
16:             return true
17:         end if
18:     end while
19:     $unsolve \leftarrow new\text{-}partt$
20:     return false
21: end function
```

In Algorithm 2, the partition algorithm is initialized and the current partition is obtained at line 5. The sound-derivation of each predicate is then generated and solved by the label constraint solver. The difference from the *One-Shot* approach is when the derived constraint is unsatisfiable: as sound, but not equivalent, derivation, it does not lead to a rejection. Instead, the predicate is recorded in *new-partt*, at line 12, to be further partitioned in the next iteration. If a derived constraint is solved, then the solution is strong enough that it does not need to be further partitioned. If the *new-partt* is empty at line 15, meaning that all the constraints in the current derivation are solved, then an early-accept solution is found. The algorithm thus accepts the constraint set at line 16. When the partition algorithm reaches the final partition and there still exists unsatisfiable constraints, the algorithm rejects the constraint set as unsatisfiable at line 20.

Iterative Early-Reject Algorithm. Early-reject algorithm makes use of complete derivations and intends to early terminate with rejection. We omit the pseudo code since it is very similar to Algorithm 2. The main difference is that it employs a complete derivation rather a sound derivation. When any label constraint is unsatisfiable, the algorithm rejects the constraint set immediately. It only accepts the input constraint set when where all the derived constraints are solved, including those from the final refinement partition.

Hybrid Algorithm. The hybrid algorithm takes advantages of both sound and complete derivations and seeks for early termination whenever possible. The pseudo code is shown in Algorithm 3.

For each predicate in the current partition, it firsts solve the sound-derivation at line 10. If the sound derivation is unsatisfiable, it solves the complete derivation at line 13 to see whether an early rejection is feasible. If not, the partition will be recorded in *new-partt* to be further partitioned. When *new-partt* is empty at line 20, early-accept is trigged. A correct implementation should never reach line 24, since the final iteration is indeed an equivalence derivation.

Algorithm 3 Hybrid Solver

```
 1: function SOLVE-EARLY-ACCEPT(cons-set)
 2:     new-partt, solution ← $\emptyset$
 3:     partition.INIT(cons-set)
 4:     while partition.STOP() == false do
 5:         partt ← partition.NEXT-PARTT(new-partt)
 6:         new-partt ← $\emptyset$
 7:         derive-sound ← SOUNDDERIVE(cons-set, partt)
 8:         derive-complete ← COMPLETEDERIVE(cons-set, partt)
 9:         for $i$ ← 0, length(partt) do
10:             if rmSolver.SOLVE(derive-sound[$i$]) then
11:                 solution[partt[$i$]] ← rmSolver.SOLUTION()
12:             else
13:                 if rmSolver.SOLVE(derive-complete[$i$]) then
14:                     unsolve ← partt[$i$]
15:                     return false
16:                 end if
17:                 new-partt.APPEND(partt[i])
18:             end if
19:         end for
20:         if new-partt == $\emptyset$ then
21:             return true
22:         end if
23:     end while
24:     return false                                        ▷ Should not reach here
25: end function
```

6 EVALUATION

We have implemented all four algorithms: One-Shot, Early-Accept, Early-Reject and Hybrid algorithms, as described in Section 5. We use Z3 [Moura and Bjørner 2008], version 4.5.1, to solve arithmetic and logical constraints (arising from predicates) and implement the Rehof-Mogensen algorithm [Rehof et al. 1999] for security-level constraints. The implementation is publicly available at https://github.com/psuplus/DerivationSolver. All experiments are run on a desktop with 16 GB of RAM and an Intel i7 processor at 2.2 GHz. During the evaluation, we were mostly interested in answering the following questions:

(1) How efficient are those algorithms in solving predicated constraints?
(2) How scalable are those algorithms in solving predicated constraints?
(3) Between sequential and combinational partitioning, which one scales better?

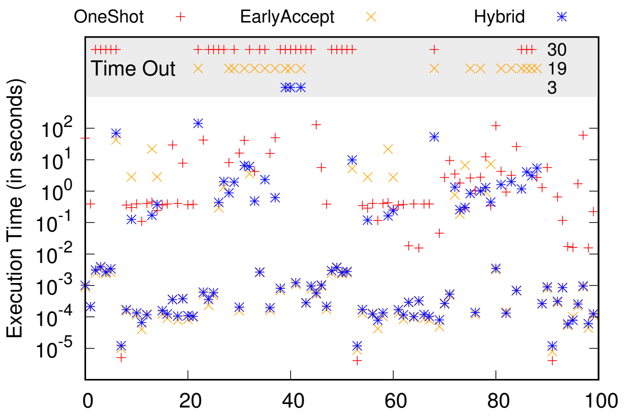

Fig. 9. Performance on the MIPS benchmark. Each algorithm times out after 180s (3 mins). The gray area shows executions that time out after 180s. The X-axis represents file ID.

6.1 Benchmarks

To evaluate constraint solving algorithms on constraints from realistic applications with information flow control, we obtain the source code of a formally verified MIPS processor with information flow control [Zhang et al. 2015]. This processor is based on a classic 5-stage in-order pipeline with separate instruction and data caches. The processor also includes typical pipelining techniques, such as data hazard detection, stalling and data bypassing.

The processor consists of 1719 lines of Verilog code, separated into 18 files. In the original version, all security labels are explicitly marked. We extend the SecVerilog compiler [Myers et al. 2015] to generate fresh label variables (to be inferred) for unannotated variables and created 50 variants mutating from the original files as follows (all removed labels are randomly picked):

- partially remove manually annotated labels, or
- split multiple modules in one file into multiple files and remove labels, making label inference more local (i.e., removing cross-module constraints), or
- remove labels and inject errors by modifying some annotated labels, or removing checks needed for security(e.g., checking b is `true` before copying a value with label $b?\mathrm{P} : \mathrm{S}$ to P). In total, 14 errors were injected to the 50 variants.

As a result, there are 2455 variables requiring security labels among all 50 files, where 1509 variables are unlabeled (i.e. to be inferred). Among the 50 files, 41 of them can only be verified with dependent labels.

The modified SecVerilog compiler generates predicated constraints in the syntax shown in Figure 5(c), which we then encode into our core language as sketched in Section 3.3. Note that since the variants of original files contain insecure code, the generated constraints tests have a mix of satisfiable and unsatisfiable constraints.

The SecVerilog compiler generates rich sets of predicates: the predicates include both path conditions and approximation of program states (details can be found in Zhang et al. [2015]). To evaluate the effect of predicates on constraint solving, we also generated predicated constraints that only contain path conditions. Our benchmark contains 100 constraint files in total: 50 set of constraints with all predicates, as well as 50 set of constraints with path conditions only.

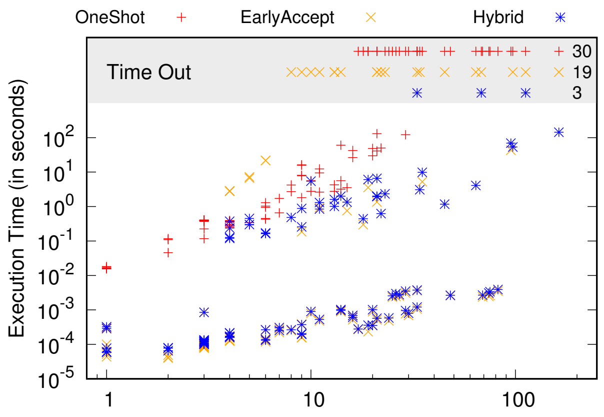

Fig. 10. Performance on the MIPS benchmark in log-log scale. The X-axis represents unique predicates involved in all constraints.

6.2 Performance of Inference Algorithms

We first compare the constraint solving time of three algorithms (one-shot, early-accept and hybrid algorithms) on the MIPS benchmark under *combinational partition*. The comparison under sequential partitioning shows similar results, so we omit that in this section. The execution time (in log scale) for each constraint file is shown in Figure 9. In the evaluation, each algorithm times out after 3 minutes. Each timed-out case is plotted in the gray area, where the number on the right counts the total number of such cases for each algorithm.

We note that in most cases, the hybrid algorithm consistently performs better than the other two algorithms: the improvement is typically by orders of magnitude compared with the one-shot algorithm, and it only times-out for 3 tests, compared with 30 tests for the one-shot algorithm. The reason is that early-termination works well in practice: for most of the constraint files, a sound and complete answer can be returned without going all the way to the exponential case as the one-shot algorithm does. The result confirms the importance of early termination. We do notice that in some hard cases (e.g., the 3 cases that the hybrid algorithm times out), all algorithms time out. For such cases, we plan to improve the hybrid algorithm as future work.

Compared with the early-accept algorithm, the hybrid algorithm has a comparable performance when both algorithms terminate. This is expected since for satisfiable constraints, the early-reject component of the hybrid algorithm has no effect. However, the hybrid algorithm only times-out for 3 tests, compared with 19 tests for the early-accept algorithm. When we inspect those 16 cases that these two algorithms differ, all of them are *unsatisfiable* constraints that require an exponential search in the early-accept algorithm. The result confirms the intuition that for a mix of satisfiable and unsatisfiable constraints, the hybrid algorithm is the most efficient.

6.3 Scalability

Figure 10 shows the execution time in terms of the number of unique predicates in the original constraints in a log-log scale (we use the number of predicates instead of the number of raw constraints since empirically, the former dominates the execution time). The hybrid method scales the best: its execution time grows slowest as the number of predicates increases (the higher plots for the hybrid algorithm suggests roughly a square-time complexity in practice). A possible reason is that most constraints can be either accepted or rejected with a small number of dependences.

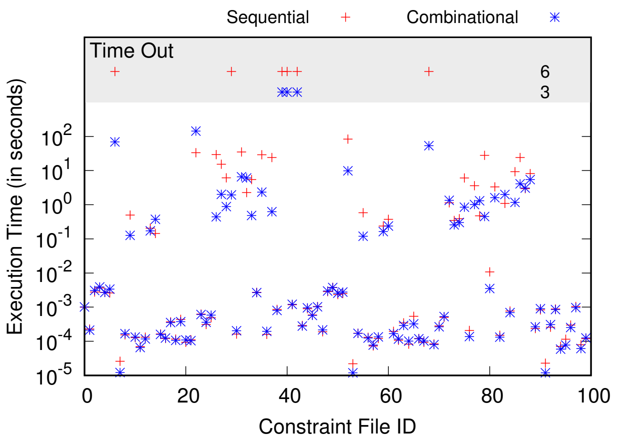

Fig. 11. Partition: Sequential vs. Combinational.

The early-accept algorithm has mixed cases: for the satisfiable ones, it scales well as the hybrid algorithm; but for the unsatisfiable ones, it not very scalable. The execution time of one-shot algorithm grows consistently in an exponential manner: it time-out for most constraints with over 20 predicates.

We note that early-accept method may grow faster than the one-shot algorithm in some cases. We believe this demonstrates the trade-off between solving a large number of small-size label constraints (the one-shot algorithm) and a small number of big-size label constraints (the early-accept algorithm). For easy cases, this trade-off inclines towards the iterative approach, but for the hard cases, as the iteration goes deeper, the size of the partition is no longer small and the accumulated efforts to generate new partitions, produce derivations and validate the derived set do not pay off. Hence, in a worse case scenario (e.g., for unsatisfiable constraints), the early-accept algorithm may scale much worse than the one-shot algorithm.

6.4 Sequential vs. Combinational Partitioning

Figure 11 shows the performance of the hybrid algorithm under different partition algorithms. Almost all test cases yield a better solution on the combinational partitioning. The cases, where two partitioning has similar performance, are mostly simple cases where a solution can be found within 10^{-2} seconds. The cases, where their performance diverges, are mostly hard cases and still combinational partitioning has a better result. This confirms the intuition that the combinational partitioning is a more stable algorithm that handles the worse case at the earliest combination level possible to avoid going to the exponential cases.

7 RELATED WORK

Security Label Inference. Typical label inference algorithms, such as those employed in Jif [Myers et al. 2006] and FlowCaml [Simonet 2003], encode the restrictions on known and unknown security levels into constraints in a finite semi-lattice. Those constraints can be solved by customized solvers (such as the Rehof-Mogensen algorithm [Rehof et al. 1999] and set-constraints solvers [Aiken

1999; Aiken et al. 1994]). However, constraints in finite semi-lattices cannot encode predicates on program states, which are essential for inferring dependent security labels.[2]

When dependent labels come to picture, various ad-hoc inference algorithms have been proposed. Lifty [Polikarpova et al. 2018] encodes information flow constraints into logical constraints (as illustrated in Section 3.3), and proposes an inference algorithm based on the inference engine of liquid types [Rondon et al. 2008; Vazou et al. 2014]. While this is a neat solution for a two-level security lattice, the encoding does not apply to applications that require multiple security labels. Recent work by Chen et al. [2018] considers security labels that depend on permissions granted to a program. Their inference algorithm is an instance of the one-shot approach: the algorithm constructs a complete partition of all (exponential) combinations of existence/absence of permissions. However, as we show in the evaluation, exploring all combinations does not scale with the increasing number of predicates in original constraints.

Vaughan and Chong [2011] propose a system for inferring declassification policies. But their work is largely orthogonal to ours: their work infers declassification policies, while our work infers dependent labels *given a security policy*.

To the best of our knowledge, no existing work offers a framework for designing and checking the soundness and completeness of dependent-label inference algorithms. Although proving a variant of one-shot algorithms as did in Chen et al. [2018] might be feasible, proving the soundness and completeness for advanced algorithms such as the hybrid algorithm can be extremely difficult without our derivation framework.

Information Flow Analysis with Dependent Security Labels. To improve the precision of static information flow analysis, dependent security labels have been introduced in various forms [Arden et al. 2012; Chen et al. 2018; Ferraiuolo et al. 2017; Li and Zhang 2017; Liu et al. 2009; Lourenço and Caires 2015; Murray et al. 2016; Myers et al. 2006; Polikarpova et al. 2018; Zhang et al. 2015]. Except for Chen et al. [2018]; Polikarpova et al. [2018], none of those works has dependent label inference; dependent security labels are annotated manually. Some prior type systems for information flow support limited forms of dependent labels [Grabowski and Beringer 2009; Jia et al. 2008; Myers 1999; Tse and Zdancewic 2007; Zheng and Myers 2007]. However, they only allow dependence on special "policy" or "label" variables, rather than run-time program states. Rich dependent type systems like Fable [Chen et al. 2010; Swamy et al. 2008], F* [Swamy et al. 2011] can encode rich information flow policies, but policy inference in those systems is arguably more challenging than within our constraint language, and hence, less likely to be fully automated.

Dependent Types. Dependent types [Martin-Löf 1982] have been widely studied and applied to practical programming languages, but most of them (e.g., Augustsson [1998]; Condit et al. [2007]; Rondon et al. [2008]; Vazou et al. [2014]; Xi [2000]; Xi and Pfenning [1999]) verifies functional correctness rather than information flow security. Though it might be possible to encode certain restricted policies into those systems (such as the encoding in Polikarpova et al. [2018]), at least there is no direct encoding for our *full constraint language* with a mix of predicates and security label constraints.

8 CONCLUSION AND FUTURE WORK

We present the first framework for designing and checking label inference algorithms for information flow analysis with dependent security labels. The framework models an inference algorithm as an iterative process where each step works on a sound and/or complete derivation of the original

[2]Conditional type [Aiken et al. 1994] integrates limited control flow information (as types) into constraints. But that work does not handle general program predicates in constraints.

constraint set. Based on the framework, we developed novel inference algorithms that are both sound and complete. Evaluation result suggests that the novel algorithms improve performance by orders of magnitude and offers better scalability compared with existing work.

There are a few directions to further improve the efficiency of our framework. First of all, it is very promising to obtain a practically efficient encoding for dynamic labels by directly adding the assumptions on dynamic labels directly into the right-hand-side constraints on security levels. Doing so requires extending the solving algorithm for label constraints (e.g., Rehof-Mogensen) to make use of the assumptions on dynamic labels, which is already implemented in Jif [Myers et al. 2006]. Second, the efficiency of the framework can be further boosted up by designing more sophisticated partitioning algorithms, such as using feedback from the solver to guide the search for predicates. A potential candidate is to use the unsatisfiable core from the current iteration to determine the predicates used in next iteration. However, the challenge is to compute a "minimum" unsatisfiable core, since a trivial core could be all constraints in the current iteration, making all predicates be added for the next iteration.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their helpful suggestions. This work was partially supported by NSF grants CCF-1566411, CNS-1702760 and CNS-1816282.

REFERENCES

Alexander Aiken. 1999. Introduction to set constraint-based program analysis. *Science of Computer Programming* 35, 2-3 (1999), 79–111.

Alexander Aiken, Edward L Wimmers, and TK Lakshman. 1994. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 163–173.

Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. 2012. Sharing Mobile Code Securely With Information Flow Control. In *Proc. IEEE Symp. on Security and Privacy (S&P)*. 191–205.

Lennart Augustsson. 1998. Cayenne—a language with dependent types. In *Proc. 3rd ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*. 239–250. http://doi.acm.org/10.1145/289423.289451

D Elliott Bell and Leonard J LaPadula. 1973. *Secure Computer Systems: mathematical foundations and model.* Technical Report M74-244. MITRE Corp., Bedford, MA.

Hongxu Chen, Alwen Tiu, Zhiwu Xu, and Yang Liu. 2018. A permission-dependent type system for secure information flow analysis. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 218–232.

Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving Compilation of End-to-end Verification of Security Enforcement. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 412–423.

Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent types for low-level programming. In *Proc. European Symposium on Programming (ESOP)*. 520–535. http://dx.doi.org/10.1007/978-3-540-71316-6_35

Jeffrey S Fenton. 1974. Memoryless subsystems. *Comput. J.* 17, 2 (1974), 143–147.

Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2017. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 555–568.

Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. In *Proc. IEEE Symp. on Security and Privacy (S&P)*. 11–20.

Robert Grabowski and Lennart Beringer. 2009. Noninterference with Dynamic Security Domains and Policies. In *Advances in Computer Science – ASIAN 2009. Information Security and Privacy*. 54–68. LNCS 5913.

Sebastian Hunt and David Sands. 2006. On Flow-sensitive Security Types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 79–90. https://doi.org/10.1145/1111037.1111045

Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. 2008. AURA: A Programming Language for Authorization and Audit. In *Proc. 13th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*. 27–38. http://doi.acm.org/10.1145/1411204.1411212

Gurvan Le Guernic and Thomas Jensen. 2005. Monitoring Information Flow. In *Workshop on Foundations of Computer Security - FCS'05*. 19–30.

Peixuan Li and Danfeng Zhang. 2017. Towards a Flow- and Path-Sensitive Information Flow Analysis. In *Proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF)*. 53–67.

Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A Platform For Secure Distributed Computation and Storage. In *Symp. on Operating Systems Principles (SOSP)*. 321–334.

Luísa Lourenço and Luís Caires. 2015. Dependent Information Flow Types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 317–328.

Per Martin-Löf. 1982. Constructive mathematics and computer programming. *Studies in Logic and the Foundations of Mathematics* 104 (1982), 153–175.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. 2016. Compositional verification and refinement of concurrent value-dependent noninterference. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*. IEEE, 417–431.

Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL)*. 228–241. http://www.cs.cornell.edu/andru/papers/popl99/popl99.pdf

Andrew C. Myers, G. Edward Suh, Danfeng Zhang, Yao Wang, , Andrew Ferraiuolo, Rui Xu, and Ian Thompson. 2015. SecVerilog 1.0: Verilog + Information Flow. (2015). Software release, http://www.cs.cornell.edu/projects/secverilog.

Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. Jif 3.0: Java Information Flow. (July 2006). Software release, http://www.cs.cornell.edu/jif.

Nadia Polikarpova, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2018. Enforcing Information Flow Policies with Type-Targeted Program Synthesis. *arXiv preprint arXiv:1607.03445v2* (2018).

François Pottier and Vincent Simonet. 2002. Information flow inference for ML. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL)*. 319–330.

Jakob Rehof et al. 1999. Tractable constraints in finite semilattices. *Science of Computer Programming* 35, 2-3 (1999), 191–221.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 159–169.

Andrei Sabelfeld and Alejandro Russo. 2010. From dynamic to static and back: Riding the roller coaster of information-flow control research. *Perspectives of Systems Informatics* (2010), 352–365.

Paritosh Shroff, Scott Smith, and Mark Thober. 2007. Dynamic dependency monitoring to secure information flow. In *Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE*. IEEE, 203–217.

Vincent Simonet. 2003. *The Flow Caml System: documentation and user's manual*. Technical Report 0282. Institut National de Recherche en Informatique et en Automatique (INRIA).

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-dependent Types. In *Proc. 16th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*. 266–278. http://doi.acm.org/10.1145/2034773.2034811

Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. 2008. Fable: A Language for Enforcing User-defined Security Policies. In *Proc. IEEE Symp. on Security and Privacy (S&P)*. 369–383. http://dx.doi.org/10.1109/SP.2008.29

Stephen Tse and Steve Zdancewic. 2007. Run-Time Principals in Information-Flow Type Systems. *ACM Trans. on Programming Languages and Systems* 30, 1 (2007), 6.

Jeffrey A Vaughan and Stephen Chong. 2011. Inference of expressive declassification policies. In *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 180–195.

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 269–282.

Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 3 (1996), 167–187.

Hongwei Xi. 2000. Imperative programming with dependent types. In *Proc. IEEE Symposium on Logic in Computer Science*. 375–387.

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*. 214–227.

Danfeng Zhang and Andrew C. Myers. 2014. Toward General Diagnosis of Static Errors. In *Proc. 14th ACM Symposium on Principles of Programming Languages (POPL)*. 569–581. http://www.cs.cornell.edu/andru/papers/diagnostic

Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proc. 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 503–516.

Lantian Zheng and Andrew C. Myers. 2007. Dynamic Security Labels and Static Information Flow Control. *Intl' J. of Information Security* 6, 2–3 (March 2007).