Benedikt Nordhoff

SECURITY THROUGH SAFETY

An Approach to Information Flow Control
Based on Derivation of Safety Properties
From a Characterisation of Insecure Behaviour

2021

Informatik

Security Through Safety

An Approach to Information Flow Control Based on Derivation of Safety Properties from a Characterisation of Insecure Behaviour

Inaugural-Dissertation
zur Erlangung des Doktorgrades
(Dr. rer. nat.)
der Naturwissenschaften im Fachbereich
Mathematik und Informatik
der Mathematisch-Naturwissenschaftlichen Fakultät
der Westfälischen Wilhelms-Universität Münster

vorgelegt von
Benedikt Heinrich Josef Nordhoff
aus Emsdetten

— 2021 —

Dekan	Prof. Dr. Xiaoyi Jiang
Erster Gutachter	Prof. Dr. Markus Müller-Olm
Zweiter Gutachter	Prof. Dr. Helmut Seidl
Tag der mündlichen Prüfung	21. Januar 2022
Tag der Promotion	21. Januar 2022

Security Through Safety

An Approach to Information Flow Control Based on Derivation of Safety Properties from a Characterisation of Insecure Behaviour

Benedikt Nordhoff

2021

Für Hanni

Abstract

Information flow control is concerned with ensuring that a program which receives confidential input does not *leak* information about this input to untrusted channels. We present a novel approach for static information flow control that can harness the power of modern safety analyses.

The approach is based on a characterisation of pairs of executions which break a security property. From the characterisation approximating safety properties are derived which ensure the security of the program. The development utilises a simple yet versatile program model that is not limited to finite control or data and targets a semantic security property which is termination insensitive but still gives some guarantees for non-terminating executions by allowing for observations throughout the execution of a program.

We provide rigorous soundness proofs that have also been machine checked and describe multiple instantiations of the approach including a fixed point–based approach targeting abstract interpretation–like safety analyses and a regular approximation that is the basis for a prototypical implementation.

Contents

1 Introduction

Data protection is a topic of broad and current interest. Multiple parties have an interest in the protection of private user data from misuse as well as its utilisation to provide services to the user and others. From a program verification perspective this leads to the problem of information flow control, where a program receives confidential input and the task is to verify that no information about this input is leaked to untrusted channels. Traditional access control as it is employed for instance in current day mobile operating systems is insufficient for this purpose, as programs often require access to both confidential data and untrusted channels to provide their desired functionality. A more sophisticated approach is therefore needed.

Less crude than the security properties enforced by access control are semantic security properties like noninterference [21], which require that whenever for two runs of a program the non-confidential inputs are equal then the untrusted outputs must also be the same. Two established approaches to verify noninterference properties for programs are security type systems [48] and slicing based on program dependency graphs [25]. Both approaches target the control structure of the program, proving that it is structured in such a way that no illicit *interference* between the confidential inputs and the untrusted outputs can happen because they are clearly separated at all times. They come in various flavours offering different degrees of precision primarily in regard to the control flow of a program. However, properties of data and their influence on the behaviour of a program, in particular which control flow paths are actually realised by executions, are mostly ignored. For this reason these approaches are still very crude over-approximations of the desired noninterference properties and cannot verify the security of programs where security is guaranteed not solely through structural means but by the overall behaviour of the program.

For a minimal example of how properties of data influence the behaviour and security of a program and how ignoring them can render a security analysis unable to verify the security of a program, consider the program snippet in Figure 1.1. Assume that the

```
if b then x = y else x = h; sec = 0 fi;
if sec > 0 then print x fi
```

Figure 1.1: A program where `h` does not interfere with **print** `x`.

variable `h` contains the confidential input and that the **print** `x` command produces output to an untrusted channel. The program uses a flag `sec` to disable output when confidential input is processed, which ensures the security of the program. Traditional approaches based on program dependence graphs or type systems will in general not be able to verify the security of programs containing such behaviour, as they do not model this influence of data properties on the control flow.

An area where the finer behaviour of programs and properties of data are more extensively studied are static program analyses for the verification of safety properties. Approaches like abstract interpretation [10, 11, 28] can exploit the power of a plethora of (numerical) abstract domains [27, 12, 22, 34, 35, 43, 15, 41, 38, 36, 30, 18] and have for instance successfully been used to verify avionics software by Airbus [13] or to validate the Monitoring and Safing Unit (MSU) of ESA's Automated Transfer Vehicle (ATV) [9, 4]. More recently in particular advances in SAT solving were harnessed for software verification based on satisfiability modulo theories (SMT) solving [6, 5].

In contrast to security properties like noninterference, which require that there are no pairs of executions that together exhibit some form of critical behaviour, safety properties require that there are no single executions that taken on their own exhibit some critical or unsafe behaviour. The verification task for safety properties is therefore to check whether the set of executions of a program is a subset of the set of safe executions. During analysis this allows at the same time to partition the space of executions and also to over-approximate the set of executions of the program. This is exploited by abstraction-based methods, which are commonly utilised for the verification of safety properties.

For security properties it is in general not possible to verify them based on a covering of the set of executions of a program. An indication for this is given by what is colloquially known as the refinement paradox [26, 2, 32], which refers to the insight that many security properties are not preserved under refinement. This is the case as a (malicious) refinement might simply resolve underspecified behaviour based on confidential inputs.

In order to harness the power behind modern safety analyses in a security context a different approach is therefore required. One approach for this are methods based on self composition [3], where the underlying safety analysis is performed on pairs of executions.

In this work we develop a different approach that instead allows to encode information about possible information flows in a given program into a safety property on single executions. The safety property guarantees the security of the program and is amenable to abstraction-based methods or can be verified directly by suitable safety analyses. We do this in a stepwise manner by first characterising pairs of executions that break a formally defined security property using a syntax driven approach that is inspired by how information flows are tracked in program dependence graphs. The characterisation is based on the simultaneous tracking of data and control dependencies within pairs of executions. We then use this characterisation to derive an approximating safety property which tracks additional dependencies, as data and control dependencies are insufficient to identify information flows within single executions. The soundness of this approach is established by rigorous soundness proofs based on a semantic security property and a simple yet flexible program model. The program model allows for infinite control and data to handle programming features like procedures, local variables or arrays. The security property is based on observations made throughout the execution of a program and corresponds to a basic noninterference property for terminating executions but additionally provides some guarantees for non-terminating executions without requiring equitermination, which tends to be overly restrictive and hard to verify for practical approaches. Finally, we describe multiple applications that employ different approaches to target the derived safety property in order to obtain practical security analyses. In particular, we describe a fixed point–based approach that can harness the power of abstract interpretation–based safety analyses. Moreover, we provide a regular approximation that can be targeted by other kinds of safety analyses and also provide some empirical results from a prototypical implementation.

Outline. The remainder of this thesis is structured as follows: Chapter 2 details some related work and also describes some motivations that went into the development of our approach. We provide definitions for the mathematical concepts used throughout this thesis in Chapter 3. In Chapter 4 we define our formal program model and the semantic security property targeted by our methodology and also provide an instantiation of the program model by a simple command language. Chapter 5 contains our central characterisation of critical pairs of executions that break our security property as well as the approximation by a safety property. The correctness proofs for our characterisation and approximation are given in Chapter 6. We describe several applications in Chapter 7 and close in Chapter 8.

2 Related Work

Security Type Systems. Security type systems for the verification of noninterference properties were introduced by Volpano et al. [48]. Security type systems work by assigning security types (or levels) to different program parts like variables, expressions and commands in a way that certain rules are respected. If a valid type can be derived for all parts of a program under these rules, the program is said to be well-typed and guaranteed to fulfil the targeted security property. One advantage of type systems is that their soundness is often comparably easy to prove, especially if paired with a semantics that is defined structurally in the same manner. They also tend to be rather declarative and succinct, as they decouple the specification of a valid type from its computation while other approaches like program dependence graphs are more algorithmic in nature. Mantel and Sudbrock [33] call type systems the "probably most popular approach to information flow analysis". There exists an abundance of variants targeting various security properties, supporting different programming features and providing varying degrees of precision. For an overview and further pointers we refer to the overview article on language-based security by Sabelfeld and Myers [42] or the work by Mantel and Sudbrock [33].

As mentioned in the introduction, security type systems usually only target the syntactic structure of a program and do not exploit properties of data as it is the goal of our approach. While core ideas of this work could most probably also be expressed in the language of type systems, they seem less suited for bridging the gap to data flow analyses like abstract interpretation–based approaches as these often do not target the syntactic structure of a program, like type systems do, but are rather defined on an execution level.

Program Dependence Graphs. Program dependence graphs (PDGs) were introduced by Ferrante et al. [16] as a tool for program optimisation and later exploited for information flow control, first by Hsieh et al. [25]. The program dependence graph of a given program is a directed graph over the set of instructions of the program. An edge within the PDG denotes a direct dependency of the sink instruction upon the source instruction, either because the sink instruction may read a value that is written by the source instruction (called a data dependency), or because the source instruction controls whether the sink instruction will be executed, e.g. the source instruction is a guarding control instruction (called a control dependency). If there exists no path in the PDG from an instruction that reads confidential input to an instruction that produces untrusted output, which is verified through slicing [50], then the program is guaranteed to fulfil a corresponding security property. PDGs and slicing algorithms come in multiple variants supporting different programming features like procedures, objects and concurrency and there exist implementations targeting prominent programming languages like Java, see for instance the works by Horwitz et al. [24], Hammer and Snelting [23] and Giffhorn and Snelting [20]. A connection to type systems is provided by Mantel and Sudbrock [33] who define a type system–based as well as a PDG-based information flow analysis for a class of multi-threaded programs and prove their equivalence.

Similar to type systems, PDGs in general only target the syntactic structure of a program and do not consider properties of data. Implementations of PDG-based security analyses like the JOANA tool by Snelting et al. [45] utilise some specific data flow analyses like points-to analyses in an initial phase to obtain a more suitable program representation that improves the handling of features like dynamic dispatch and aliasing before beginning with the actual PDG-based analysis.

Our methodology is inspired by the way PDGs track information flows through data and control dependencies. Instead of a graph on the set of instructions of a program our approach defines dependencies between points in the executions of the program and based on these it provides a characterisation of executions violating a security property. A similar concept of dynamic dependencies, which are defined for paths in the control flow graph, can be found in the soundness proofs for PDG-based approaches by Wasserrab et al. [49] and Giffhorn and Snelting [20]. However, other than through the PDG these dependencies are not clearly connected to program executions breaking the security property, which is a central feature of our approach.

In Section 7.1 we provide a direct comparison to the PDG-based approach by defining PDGs for our program model as a simple instantiation of our approach and utilise our results to directly derive a soundness property. In Section 7.3.2 we also prove for another instantiation of our approach that it is as least as precise as the PDG-based approach.

Path Conditions. Path conditions based on program dependence graphs are one attempt to exploit how data properties influence the control flow of a program in order to improve information flow analysis. Snelting [44] and Robschink and Snelting [40] define a path condition as a necessary condition for information flow along a PDG path. Work on this focused on how to efficiently compute such conditions by collecting control flow predicates that dominate instructions on a PDG path. The calculated path condition was then checked by an SMT-solver with the goal to prove it unsatisfiable and deduce the security of the program. The soundness of this approach has however not been justified rigorously. Taghdiri et al. [46] provide the intuitive justification that path conditions are exploiting the fact that flow can only happen along PDG paths.

Path conditions did in fact motivate the development in this thesis. An initial idea was to improve path conditions by combining them with global program invariants obtainable from data flow analyses, as path conditions themselves do not consider how the values of program variables occurring in the collected control predicates are computed. This however resulted in soundness problems as path conditions obtained from insecure programs turned out to be unsatisfiable when combined with valid program invariants.

Unlike path conditions our approach is built upon a rigorous soundness proof that has also been machine checked. Instead of generating a verification condition to be checked by a constraint solver based on some aspects extracted from a PDG path our approach directly characterises executions that violate a security property and thereby allows the employed data flow analyses to exploit arbitrary aspects that play into the feasibility of these executions.

Self Composition. The potential benefits of reducing security problems to safety problems has also been studied by several authors through means of self composition. A self composition of a program is a new program that contains two independent copies of the original program such that the executions of the self composition correspond exactly to pairs of executions of the original program. This allows one to express security properties like noninterference of the original program as a safety property of the self composition. Barthe et al. [3] define a simple self composition and provide examples of how to exploit it to verify security using Hoare logic, a weakest precondition calculus, separation logic as well as temporal logic. Terauchi and Aiken [47] observe that practical safety analyses, in their case the BLAST model checker, are not able to verify the required safety properties when using naive self compositions and propose a type-directed self composition inspired by security type systems. Kovács et al. [29] and Müller et al. [37] also explore more sophisticated approaches to construct adequate self compositions and employ relational abstract interpretations to verify the required safety properties. Conceptually similar approaches can also be found in the field of regression verification, which aims at checking two programs for equivalence, e.g. by Felsing et al. [14] who employ a weakest precondition calculus based on a composition of the programs in order to generate verification conditions in the form of Horn clauses to be checked by an SMT solver.

Self composition–based approaches still require the employed safety analyses to essentially handle two executions at once and verify that they produce the same outputs and therefore rely on more powerful relational safety analyses. Our approach instead allows to consider only single executions of the original program and encodes information about possible influences into a simpler reachability problem which might also be checked with cheaper non-relational safety analyses. A potential connection might be found in our central characterisation of critical execution pairs. While not explored further in this work, it might be possible to exploit this to also define an optimised self composition that only has to consider a reduced set of execution pairs.

3 Preliminaries

Before we begin with our formal development in the next chapter we record some general definitions and conventions used throughout the rest of this work.

Fundamentals. Theorems and other statements will in general contain free variables, which are to be considered universally quantified over the greatest appropriate domain, such that all appearing terms are well-defined. In the same way we might omit the domains in explicit existential or universal quantifiers where also the greatest appropriate domain is to be assumed.

Quantifiers ($\exists, \forall, \nexists$ (a shorthand for $\neg\exists$)) have the lowest precedence of all operators and extend to the end of the innermost scope in which they were introduced. They are followed with increasing precedence by: logical implications ($\Rightarrow, \Leftarrow, \Leftrightarrow$), disjunctions ($\vee$), conjunctions ($\wedge$), negation ($\neg$), relational operators (e.g. $\leq$, $\xrightarrow{pd}$, $\xrightarrow{cd}$), binary operators (e.g. $+$), unary operators and function applications.

Intervals, Sequences and Relations. We denote by $\mathbb{N}$ the set of natural numbers including zero. Equations (3.1) to (3.4) define closed, open and half-open intervals of natural numbers, denoted by $[i,j]$, (i,j), $[i,j)$, $(i,j]$ for $i \in \mathbb{N}$ and $j \in \mathbb{N} \cup \{\infty\}$ where for right open intervals the upper bound j might be infinite.

$$[i,j] = \{n \in \mathbb{N} \mid i \leq n \leq j\} \tag{3.1}$$

$$(i,j) = \{n \in \mathbb{N} \mid i < n < j\} \tag{3.2}$$

$$[i,j) = \{n \in \mathbb{N} \mid i \leq n < j\} \tag{3.3}$$

$$(i,j] = \{n \in \mathbb{N} \mid i < n \leq j\} \tag{3.4}$$

We utilise finite and infinite sequences which we represent as families indexed by natural numbers starting at zero and denote as $(n_i)_{i<k}$ where $k \in \mathbb{N} \cup \{\infty\}$. The

empty sequence $(n_i)_{i<0}$ is denoted by ϵ and we identify single elements with sequences of length one. We define the prefix order on finite and infinite sequences by $(n_i)_{i<k} \leq (n'_i)_{i<k'} \Leftrightarrow k \leq k' \wedge \forall i < k \colon n_i = n'_i$. For any sequence $\pi = (n_i)_{i<k}$ and any $l < k$ we denote by $\pi_l = (n_i)_{i\leq l}$ the prefix of π up to l. Conversely we define a shift operator $(\lll)$ for sequences by $(n_i)_{i<k} \lll l = (n_{i+l})_{i<k-l}$ with the convention that $\infty - l = \infty = \infty + l$ for any $l \in \mathbb{N}$. Moreover we define the concatenation operator $(\cdot)$ for sequences by $(n_i)_{i<k} \cdot (n'_i)_{i<k'} = (m_i)_{i<k+k'}$ where $m_i = n_i$ for $i < k$ and $m_i = n'_{i-k}$ for $i \geq k$. Note that this definition allows the first sequence to be infinite in which case it returns the first sequence. The concatenation operator is lifted to sets of sequences via $X \cdot Y = \{x \cdot y \mid x \in X \wedge y \in Y\}$. For any set X and natural number $k \in \mathbb{N}$ we denote by $X^k = \{(x_i)_{i<k} \mid \forall i < k \colon x_i \in X\}$ the set of sequences of length k over X, by $X^* = \bigcup_{k\geq 0} X^k$ the set of all finite sequences over X, by $X^+ = \bigcup_{k\geq 1} X^k$ the set of all finite non-empty sequences over X and by $X^{\leq\omega}$ the set of all sequences over X, finite or infinite.

For any endorelation $E \subseteq X \times X$ we overload the notation defined above and let $E^+ = \bigcup_{k\geq 1} E^k$ denote the transitive and $E^* = \bigcup_{k\geq 0} E^k$ the transitive reflexive closure, inductively defined by $E^0 = \{(x,x) \mid x \in X\}$ and $E^{k+1} = \{(x,z) \mid \exists y \colon (x,y) \in E^k \wedge (y,z) \in E\}$. For any binary relation $R \subseteq X \times Y$ we write $x \; R \; y$ to denote $(x,y) \in R$ and also chain this notation, e.g. $x \; E \; y \; R \; z$ for $(x,y) \in E \wedge (y,z) \in R$.

Finite Automata. We utilise finite automata to represent regular sets of finite sequences. A finite automaton is a five-tuple $\mathcal{A} = (Q, N, \delta, q_0, Q_f)$ where Q is the finite set of states, $q_0 \in Q$ is the initial state, $Q_f \subseteq Q$ is the set of accepting states, the alphabet N is a finite set of symbols over which the automaton operates and $\delta \subseteq Q \times N \times Q$ is the transition relation. We write $q \xrightarrow{n}_\delta q'$ to denote $(q, n, q') \in \delta$ as well as $q \xrightarrow{(n_i)_{i<k}}_\delta q'$ for sequences $(n_i)_{i<k} \in N^*$ to denote $\exists (q_i)_{i\leq k} \in Q^{k+1} \colon q_0 = q \wedge q_k = q' \wedge \forall i < k \colon q_i \xrightarrow{n_i}_\delta q_{i+1}$. The language $\mathcal{L}(\mathcal{A}) \subseteq N^*$ of an automaton is defined by $\mathcal{L}(\mathcal{A}) = \bigcup_{q\in Q_f} \mathcal{L}_\mathcal{A}(q)$ where $\mathcal{L}_\mathcal{A}(q) = \{\pi \in N^* \mid q_0 \xrightarrow{\pi}_\delta q\}$ is the language accepted by q in $\mathcal{A}$.

Functions. For any mapping $f \colon X \to Y$ we denote its domain X by $\mathrm{dom}(f)$ and by $\mathrm{ran}(f) = \{f(x) \mid x \in X\}$ its range and for any subset of its domain $Z \subseteq X$ we denote by $f\restriction_Z \colon Z \to Y$ the restriction of f to the domain Z. For any set Z we define for maps whose domain encompasses Z the pointwise equivalence on Z by $f =_Z g \Leftrightarrow \forall x \in Z \colon f(x) = g(x)$. We utilise $\mapsto$ to define anonymous functions, where $x \mapsto t(x)$ denotes the function mapping any x from an appropriate domain, which as for quantifiers will be clear from the context, to $t(x)$. For any function $f\colon X \to Y$, $x \in X$ and $y \in Y$ we define the function update of f with y for x denoted by $f\{y/x\}$ as the function from X to Y that maps x to y and all other elements in the same way as f and extend this to updates of multiple variables by defining $f\{y_1/x_1, \dots, y_n/x_n\} = f\{y_1/x_1\} \dots \{y_n/x_n\}$. For any endofunction $f\colon X \to X$ and $i \in \mathbb{N}$, $f^i\colon X \to X$ denotes the function applying f repeatedly i times.

Orders, Lattices, Monotone Functions and Fixed Points. A partial order is a tuple $(D, \sqsubseteq)$ consisting of a domain D and relation $\sqsubseteq \subseteq D \times D$ which is reflexive, that is for all $a \in D$ it holds that $a \sqsubseteq a$, transitive, that is for all $a, b, c \in D$ it holds that if $a \sqsubseteq b$ and $b \sqsubseteq c$ then it must also hold that $a \sqsubseteq c$, as well as antisymmetric, that is for all $a, b \in D$ it holds that if $a \sqsubseteq b$ and $b \sqsubseteq a$ then it must already be the case that $a = b$.

An endofunction $f\colon D \to D$ on a partial order $(D, \sqsubseteq)$ is called monotone, if for all $a, b \in D$ it holds that if $a \sqsubseteq b$ then it also holds that $f(a) \sqsubseteq f(b)$. Moreover a function $f\colon X_1 \times \cdots \times X_n \to D$ with $X_i = D$ is called monotone in the i-th argument, if for all $x_1 \in X_1, \dots, x_{i-1} \in X_{i-1}, x_{i+1} \in X_{i+1}, \dots, x_n \in X_n$ the function $d \mapsto f(x_1, \dots, x_{i-1}, d, x_{i+1}, \dots, x_n)$ is monotone. An element $d \in D$ is called a pre-fixed point of an endofunction $f\colon D \to D$, if $f(d) \sqsubseteq d$ and is called a fixed point if $f(d) = d$.

An element $d \in D$ is called an upper bound for a set of elements $D' \subseteq D$, if for all $d' \in D'$ it holds that $d' \sqsubseteq d$ and is called a lower bound for D', if for all $d' \in D'$ it holds that $d \sqsubseteq d'$. A partial order is called a complete lattice, if for any subset $D' \subseteq D$ there exists a unique least upper bound $\bigsqcup D' \in D$ (also called the join or supremum), that is an upper bound which is also a lower bound for the set of all upper bounds of the subset, as well as a greatest lower bound $\bigsqcap D' \in D$ (also called the meet or infimum), that is a lower bound which is also an upper bound for the set of all lower bounds of the subset. For a complete lattice the least element, which exists as it is the least upper bound of the empty set, is denoted by $\bot$ and the greatest element, which exists as it is the greatest lower bound of the empty set, is denoted by $\top$. By $\mathbb{B}$ we denote the complete lattice of two elements $\{\bot, \top\}$, where $\top$ represents truth and a statement consisting of a $\mathbb{B}$ valued term t denotes $t = \top$.

For any set X and complete lattice $(D, \sqsubseteq)$ the set of mappings from X to D with the pointwise order forms a complete lattice $(X \to D, \sqsubseteq)$ where $f \sqsubseteq g \Leftrightarrow \forall x \in X\colon f(x) \sqsubseteq g(x)$. For a mapping $f\colon X \to D$ and $x \in X$ we denote by $f\lfloor_x\colon X \to D$ the mapping that maps x to $f(x)$ and all other elements to the bottom element of D. For an endofunction $f\colon D \to D$ on a complete lattice D we denote by $f^*\colon D \to D$ the function $d \mapsto \bigsqcup\{f^i(d) \mid i \geq 0\}$ and by $f^+\colon D \to D$ the function $d \mapsto \bigsqcup\{f^i(d) \mid i \geq 1\}$.

Graphs, Paths and Reachability. A directed graph is a tuple (N, E) where N is an arbitrary set of so called nodes and $E \subseteq N \times N$ is a set of so called edges. A path in a directed graph (N, E) is a sequence $(n_i)_{i<k} \in N^{\leq\omega}$ of nodes, either finite or infinite, such that for all $i \in (0, k)$ it holds that $(n_{i-1}, n_i) \in E$. A finite path $(n_i)_{i\leq k}$ is said to lead from a node n to a node m, if $n_0 = n$ and $n_k = m$ and we say that the path reaches m. An infinite path $(n_i)_{i>0}$ is said to reach a node n, if there exists an index k, such that $n_i = n$ for all $i \geq k$. In the context of a path $(n_i)_{i<k}$ we say that an index i, or the path at i, reaches a node n if $n_i = n$. A node m is said to be reachable from a node n, if there exists a path leading from n to m and in this case we also say that n reaches m. A rooted graph is a tuple $(N, E, \mathfrak{st})$ where $\mathfrak{st} \in N$ and (N, E) is a directed graph. A path $(n_i)_{i<k}$ in a rooted graph is said to be initial if $n_0 = \mathfrak{st}$. A node n is said to be reachable in a rooted graph $(N, E, \mathfrak{st})$ if n is reachable from $\mathfrak{st}$.

4 Program Model and Security Property

We now move to defining the formal security property as well as the program model that we will use throughout this work. We also give an exemplary instantiation of our abstract program model by a command language that we will use for examples.

4.1 Security Property

Our security property is a noninterference-like property where sufficiently similar inputs are required to produce sufficiently similar observable behaviours for a given attacker. We therefore require notions of input, similarity of inputs, produced observable behaviour and sufficient similarity of behaviours. The precise structure of inputs, their similarity and the observable behaviour it produces will be defined when we fix our program model. For our definition of security we initially assume that there is an arbitrary set Σ of possible inputs together with an equivalence relation $=_L \subseteq \Sigma \times \Sigma$, relating those inputs that should lead to similar behaviour for the attacker. The equivalence relation is suggestively named $=_L$ or low equivalence, as our program model will handle inputs through values of variables in initial states and the similarity relation will relate initial states that coincide on a set L of variables, whose values are allowed to influence the observable behaviour. For any input the program model will define a sequence of values that can be observed throughout the execution of the program, which we assume at this point to be given directly by a function mapping inputs to sequences of observed values. In order to define when two observable behaviours are sufficiently similar we additionally rely on a notion of termination, here given in form of a set of inputs for which the program terminates. We do so as requiring non-terminating executions to produce exactly the same observations would demand analyses to prove equitermination of certain loops, which is either rather challenging or results in overly restrictive approximations. Our security property therefore only requires terminating executions to produce exactly the same sequences of observations and allows the observation sequence produced by a non-terminating execution to be truncated at some point and therefore only requires that one sequence of observations is a prefix of the other for general executions. As terminating executions must not drop any observations, we require that for any input, independent of whether the execution terminates or not, the corresponding observation sequence must be a prefix of any observation sequence that is produced by a sufficiently similar input for which the execution terminates.

Definition 4.1 (Security)**.** An observation function $\mathrm{obs}\colon \Sigma \to V^{\leq\omega}$ is called *secure in regard to an equivalence relation $=_L \subseteq \Sigma \times \Sigma$ and termination predicate $T \subseteq \Sigma$* if and only if the following statements hold:

$$\forall \sigma \in T\colon \forall \sigma' \in \Sigma\colon \sigma =_L \sigma' \Rightarrow \mathrm{obs}(\sigma') \leq \mathrm{obs}(\sigma) \tag{4.1}$$

$$\forall \sigma, \sigma' \in \Sigma\colon \sigma =_L \sigma' \Rightarrow \mathrm{obs}(\sigma) \leq \mathrm{obs}(\sigma') \vee \mathrm{obs}(\sigma) \geq \mathrm{obs}(\sigma') \tag{4.2}$$

While the definition does not directly consider pairs of terminating executions, due to the fact that the low equivalence relation $=_L$ is symmetric and the prefix ordering on sequences $\leq$ is antisymmetric, equation (4.1) directly yields that terminating executions on low equivalent input states have to produce the same sequences of observations, which we record in the following lemma.

Lemma 4.1. *For a secure observation function* $\mathrm{obs}\colon \Sigma \to V^{\leq\omega}$ *in regard to* $=_L \subseteq \Sigma \times \Sigma$ *and* $T \subseteq \Sigma$ *it holds that* $\forall \sigma, \sigma' \in T\colon \sigma =_L \sigma' \Rightarrow \mathrm{obs}(\sigma) = \mathrm{obs}(\sigma')$. □

Note that this notion of security corresponds to the notion of indistinguishable security as used by Bohannon et al. [8].

In comparison with classical noninterference properties on input/output states making observations throughout the run of a program allows us to also obtain some security guarantees for non-terminating executions, which are both more lenient and expressive than requiring equitermination and allow for more relaxed and precise approximations for analysis.

4.2 Program Model

While our definition of security is based on a direct mapping from inputs to observable behaviour, the program model defined in this section will make it explicit how those behaviours are produced and contains all necessary components and assumptions upon which we will base our further development in the next chapters.

The program model is based on control flow graphs that lend themselves straight forwardly to the application of e.g. abstract interpretation–based safety analyses and are widely used both throughout the safety and security community. We use a deterministic semantic because we already opted for deterministic behaviour in our definition of security and security properties for non-deterministic programs tend to be either very restrictive or hard to verify and are often not stable under refinement. Our control flow graphs are node annotated with state transformers associated to control locations, such that the semantic is given by an endofunction on location-state pairs. States are maps from an arbitrary set of variables to an arbitrary set of values. We allow for arbitrary state transformers that have to fulfil some semantic assumptions in order to be consistent with the abstractions fixed by the model. This allows for our abstract program model to be instantiated for a variety of concrete models, which we will illustrate by instantiating it for a simple command language in the next section. Note that neither the set of control locations nor the set of variables or values are assumed to be finite, which allows the model to also be instantiated in infinite state settings e.g. with procedural programs or arrays.

Definition 4.2 (Program model)**.** An *admissible program (with control and data abstractions and security specification)* is a decuple $(\Sigma, N, E, \mathfrak{st}, \mathfrak{te}, [\![.]\!], \mathrm{def}, \mathrm{use}, L, \mathrm{obs})$ where:

- $\Sigma = \mathrm{Var} \to \mathrm{Val}$ is the set of *states* mapping *variables* from an arbitrary set of variables Var to *values* from Val,
- N is an arbitrary set of *control locations*,
- $\mathfrak{st} \in N$ is the *initial control location*,
- $\mathfrak{te} \in N$ is the *terminal control location*,
- $[\![.]\!] \colon N \times \Sigma \to N \times \Sigma$ is the deterministic *semantic transfer function*,
- $E \subseteq N \times N$ is the *control flow abstraction*,
- $\mathrm{def} \colon N \to 2^{\mathrm{Var}}$ is the *abstraction of variables written at control locations*,

- use: $N \to 2^{\mathrm{Var}}$ is the *abstraction of variables read at control locations*,
- $L \subseteq \mathrm{Var}$ is the set of *low input variables* and
- obs: $X \to (\Sigma \to \mathrm{Val})$ for some $X \subseteq N$ is the *attacker model.*

We denote the projections of $[\![.]\!]$ to locations N and states Σ by $[\![.]\!]_l \colon N \times \Sigma \to N$ and $[\![.]\!]_s \colon N \times \Sigma \to \Sigma$, which satisfy $\forall \sigma, n \colon [\![n, \sigma]\!] = ([\![n, \sigma]\!]_l, [\![n, \sigma]\!]_s)$. For a program to be admissible the following well-formedness assumptions have be fulfilled:

1. The terminal control location $\mathfrak{te}$ only admits identity self loops:
$$\forall \sigma \in \Sigma \colon [\![\mathfrak{te}, \sigma]\!] = (\mathfrak{te}, \sigma).$$
2. The terminal control location is reachable from any control location according to the control flow abstraction:
$$N \times \{\mathfrak{te}\} \subseteq E^*.$$
3. The control flow abstraction safely approximates the semantic control flow:
$$\forall n \in N \colon \forall \sigma \in \Sigma \colon (n, [\![n, \sigma]\!]_l) \in E.$$
4. The control flow abstraction models the terminal self loop precisely:
$$(\{\mathfrak{te}\} \times N) \cap E = \{(\mathfrak{te}, \mathfrak{te})\}.$$
5. The write abstraction captures the variables modified at control locations:
$$\forall n \in N \colon \mathrm{def}(n) \supseteq \{v \in \mathrm{Var} \mid \exists \sigma \colon \sigma(v) \neq [\![n, \sigma]\!]_s(v)\}.$$
6. The read abstraction safely captures the variables that determine the next control location, the values of the written variables and a possible observation, that is $\forall n \in N \colon \forall \sigma, \sigma' \in \Sigma \colon$
$$\sigma =_{\mathrm{use}(n)} \sigma' \Rightarrow \begin{cases} [\![n, \sigma]\!]_l = [\![n, \sigma']\!]_l \land \\ [\![n, \sigma]\!]_s =_{\mathrm{def}(n)} [\![n, \sigma']\!]_s \land \\ (n \in \mathrm{dom}(\mathrm{obs}) \Rightarrow \mathrm{obs}(n)(\sigma) = \mathrm{obs}(n)(\sigma')). \end{cases}$$

7. The read and write abstractions map the initial control location to the non-low variables:
$$\text{def}(\mathfrak{st}) = \text{use}(\mathfrak{st}) = \text{Var} \setminus L.$$

While the behaviour and security of a program will be defined solely based on L, obs, Σ, $\mathfrak{st}$, $\mathfrak{te}$ and $[\![.]\!]$, this definition also fixes data and control abstractions with the necessary assumptions, upon which our methodology relies. The behaviour of a program is defined through the transfer function $[\![.]\!]$, which is repeatedly applied starting with the initial control location $\mathfrak{st}$ and a given input state from Σ until the terminal control location $\mathfrak{te}$ is reached, at which point the semantic only performs an identity self loop according to Assumption 1. This simplifies the handling of terminating and non-terminating executions by avoiding any structural differences as all executions are always infinite as defined in the following definition.

Definition 4.3 (Execution)**.** Given an admissible program (Σ, N, E, $\mathfrak{st}$, $\mathfrak{te}$, $[\![.]\!]$, def, use, L, obs) for any $\sigma \in \Sigma$ we call $([\![\mathfrak{st}, \sigma]\!]^i)_{0 \le i}$ the *corresponding execution.* An execution $([\![\mathfrak{st}, \sigma]\!]^i)_{0 \le i}$ is called *terminating* if there exists an index $i \geq 0$ and state $\sigma' \in \Sigma$ such that $[\![\mathfrak{st}, \sigma]\!]^i = (\mathfrak{te}, \sigma')$.

For simplicity we introduce the convention that in any context where we fix an initial state denoted by $\sigma \in \Sigma$ we implicitly also fix $(n_i)_{0 \le i}$ and $(\sigma_i)_{0 \le i}$ to denote the elements of the corresponding execution that is we have $([\![\mathfrak{st}, \sigma]\!]^i)_{0 \le i} = ((n_i, \sigma_i))_{0 \le i}$. In the same way we fix $(n'_i)_{0 \le i}$ and $(\sigma'_i)_{0 \le i}$ whenever we fix an initial state denoted by σ'.

Besides the behaviour of the program itself in the form of its executions, Definition 4.2 also fixes our model of an attacker. The behaviour an attacker might observe during the execution of a program is given by observations that occur whenever a control location from dom(obs) is visited by an execution and allow the attacker to observe the value produced by evaluating the mapping annotated to the control location by obs in the reached state. The resulting sequence of values is the behaviour, in the sense of Definition 4.1, observed by the attacker. Note that we do not model the fact that observations within an execution might occur at different times in an actual execution, which an attacker might also observe, our security property is in this sense timing insensitive.

Definition 4.4 (Observable behaviour)**.** Given an admissible program $(\Sigma, N, E, \mathfrak{st}, \mathfrak{te}, [\![.]\!], \mathrm{def}, \mathrm{use}, L, \mathrm{obs})$, the *corresponding observation function* as required by Definition 4.1 is defined by lifting obs to Σ through $\mathrm{obs}(\sigma) = (\mathrm{obs}(n_{i_j})(\sigma_{i_j}))_{j<k}$ where $(i_j)_{j<k}$ with $k \in \mathbb{N} \cup \{\infty\}$ is the maximal strictly increasing family of indices such that $n_{i_j} \in \mathrm{dom}(\mathrm{obs})$ for all j less than k.

In addition to the attacker model, which defines the observable behaviour, our security property requires a model of confidential information that the attacker must not observe and a termination predicate. We already defined the latter in the definition of executions, where an execution is considered terminating if it reaches the terminal control location $\mathfrak{te}$. The confidential information is defined through an equivalence relation on the input states, which our model defines through the set of *low* variables L, whose values in an initial state are allowed to influence the behaviour the attacker can observe.

Definition 4.5 (Security of a program)**.** An admissible program $(\Sigma, N, E, \mathfrak{st}, \mathfrak{te}, [\![.]\!]$, def, use, L, obs) is called *secure* if the observation function obtained by lifting obs to Σ as in Definition 4.4 is secure in regard to $=_L$ and $T = \{\sigma \in \Sigma \mid \exists i\colon n_i = \mathfrak{te}\}$ according to Definition 4.1.

The remaining components fixed by Definition 4.2, namely the control flow abstraction E, the abstractions of variables read and written at nodes (use and def) and the corresponding Assumptions 2 to 7, are not required to define the behaviour or security of a program itself but build the connection point for our characterisations of information flows in Chapter 5 and are the basis for further abstractions that we will perform from there to obtain sound and effective analysis methods for our security property.

Before we move to defining our central characterisation of information flows in the next chapter we give an exemplary instantiation of our abstract model in the following section in order to assure ourselves of its adequacy and enable us to present concise and well-defined examples later on.

4.3 Command Language

While the abstract program model we defined in the previous section will underlie the formal development in this work we will now give a concrete instantiation in the form of a simple command language that we will use for examples.

The syntax of our command language is parameterised by a set of variable names Var, a set of constant names $\mathcal{CT}$, which we assume to include the constant $0 \in \mathcal{CT}$, and a set of binary operator names $\mathcal{OP}$. Expressions to be used for assignments and control flow guards are constructed inductively from those. We fix the sets Var, $\mathcal{CT}$, $\mathcal{OP}$ for the remainder of this section. We use abstract grammars to define the expressions and commands of our command language. While we use textual representations to denote commands and expressions, the actual objects these represent are abstract terms that possess unique productions in our abstract grammars. To this end we do not include parenthesis or precedence rules in our grammars but might use them in textual representations to clarify which abstract term is denoted.

Definition 4.6 (Expressions)**.** The set of *expressions* $\mathcal{E}$ is a term language defined by the grammar

$$e ::= v \mid c \mid e \; op \; e \quad \text{for } v \in \mathrm{Var},\ c \in \mathcal{CT},\ op \in \mathcal{OP}.$$

In order to establish the semantics for expressions we fix a set of values Val, with which the set of states Σ is given as the set of all maps from the set of variables to the set of values ($\Sigma = \mathrm{Var} \to \mathrm{Val}$). The semantics is then determined by a given evaluation function for constants $\mathrm{eval_c}\colon \mathcal{CT} \to \mathrm{Val}$, mapping constants to values and a given evaluation function for operators $\mathrm{eval_{op}}\colon \mathcal{OP} \times \mathrm{Val} \times \mathrm{Val} \to \mathrm{Val}$, combining values for operators.

Definition 4.7 (Expression semantics)**.** The *semantics for expressions* is given through the *evaluation function* $\mathrm{eval}\colon \mathcal{E} \times \Sigma \to \mathrm{Val}$, defined recursively over the structure of the first argument by

$$\mathrm{eval}(e, \sigma) = \begin{cases} \sigma(v) & \text{if } e = v \in \mathrm{Var}, \\ \mathrm{eval_c}(c) & \text{if } e = c \in \mathcal{CT}, \\ \mathrm{eval_{op}}(op, \mathrm{eval}(e_1, \sigma), \mathrm{eval}(e_2, \sigma)) & \text{if } e = e_1 \; op \; e_2. \end{cases}$$

For a concrete instantiation we define the set of arithmetic expressions over alphabetical variables and integer values in Example 4.1, which we will use as the default set of expressions in later examples.

Example 4.1 (Arithmetic expressions)**.** The set of *arithmetic expressions* uses alphabetical strings for variables $\mathrm{Var} = \{a, \dots, z\}^+$, integers as constants $\mathcal{CT} = \mathbb{Z}$ and the operators $\mathcal{OP} = \{+, -, *, <\}$. It uses integers as values $\mathrm{Val} = \mathbb{Z}$, such that constants are evaluated to themselves via $\mathrm{eval_c}(i) = i$ and operators $(+, -, *, <)$ are interpreted as the usual arithmetic operators that is

$$\mathrm{eval_{op}}(op, i, j) = \begin{cases} i + j & \text{if } op = +, \\ i - j & \text{if } op = -, \\ i * j & \text{if } op = *, \\ 1 & \text{if } op = < \wedge\, i < j, \\ 0 & \text{if } op = < \wedge\, i \geq j. \end{cases}$$

Based on any set of expressions we define the corresponding set of commands in the form of a simple while-language. The language contains the basic `skip` command which does not alter the state, assignments of the form $v = e$ which allow the result of evaluating expressions in the current state to be assigned to a variable in the successor state, *if-then-else* statements which allow branching based on the value of an expression in the current state, where, if the value is zero, the *else* branch is taken and otherwise the *then* branch is taken, *while* statements, which repeatedly execute a command until the guarding expression evaluates to zero, and *print* statements, which output the value of an expression visibly to an attacker and are used to define the observations made by an attacker during the execution of one of our programs, as well as sequential compositions of commands.

Definition 4.8 (Commands)**.** The set of *commands* $\mathcal{C}$ over an expression set $\mathcal{E}$ over $(\mathrm{Var}, \mathcal{CT}, \mathcal{OP})$ is defined as the set of all terms derivable by the following grammar:

$$\begin{aligned} C &::= \texttt{skip} \mid v = e \mid \texttt{if } e \texttt{ then } S \texttt{ else } S \texttt{ fi} \mid \texttt{while } e \texttt{ do } S \texttt{ od} \mid \texttt{print } e \\ &\qquad \text{for } e \in \mathcal{E}, v \in \mathrm{Var} \\ S &::= C \mid C; S \end{aligned}$$

Note that by definition the left-hand side (C) of a sequential composition $(C; S)$ can not be a sequential composition itself. We nonetheless write $C_1; C_2$ for arbitrary C_1, C_2

to denote the command where the elements of C_1 have successively been composed with C_2 by recursively unfolding $(C; S); C_2$ to $C; (S; C_2)$. Our default set of commands will be the set of those over the set of arithmetic expressions with alphabetical variables as defined in Example 4.1.

We define a small step semantics as an endofunction on pairs of commands and states and use the singular `skip` command with an identity self loop to model termination as we did in Definition 4.2.

Definition 4.9 (Command semantics)**.** The *semantics* $[\![.]\!]\colon \mathcal{C} \times \Sigma \to \mathcal{C} \times \Sigma$ for a set $\mathcal{C}$ of commands over a set of expressions $\mathcal{E}$ with expression semantics $\mathrm{eval}\colon \mathcal{E} \times \Sigma \to \mathrm{Val}$ is defined recursively by:

$$[\![\texttt{skip}, \sigma]\!] = (\texttt{skip}, \sigma) \qquad [\![v = e, \sigma]\!] = (\texttt{skip}, \sigma\{\mathrm{eval}(e, \sigma)/v\})$$

$$[\![\texttt{print}\ e, \sigma]\!] = (\texttt{skip}, \sigma)$$

$$[\![\texttt{if}\ e\ \texttt{then}\ C_t\ \texttt{else}\ C_f\ \texttt{fi}, \sigma]\!] = \begin{cases} (C_f, \sigma) & \text{if } \mathrm{eval}(e, \sigma) = \mathrm{eval}(0, \sigma) \\ (C_t, \sigma) & \text{otherwise} \end{cases}$$

$$[\![\texttt{while}\ e\ \texttt{do}\ C\ \texttt{od}, \sigma]\!] = \begin{cases} (\texttt{skip}, \sigma) & \text{if } \mathrm{eval}(e, \sigma) = \mathrm{eval}(0, \sigma) \\ (C; \texttt{while}\ e\ \texttt{do}\ C\ \texttt{od}, \sigma) & \text{otherwise} \end{cases}$$

$$[\![C_1; C_2, \sigma]\!] = \begin{cases} (C_2, \sigma') & \text{if } [\![C_1, \sigma]\!] = (\texttt{skip}, \sigma') \\ (C_1'; C_2, \sigma') & \text{if } [\![C_1, \sigma]\!] = (C_1', \sigma') \wedge C_1' \neq \texttt{skip} \end{cases}$$

We note two properties of these semantics. Firstly `print` commands are treated within the semantics just like `skip` commands, they will only obtain further meaning when we define the attacker later on. Secondly, the way we treat sequential compositions and `skip` commands means that when an `if` command contains a simple `skip` in a branch, taking that branch directly skips over this command and moves on to the next command when the `if` command is part of a sequential composition. We therefore omit the `skip` command in some cases and write `if` e `then` C `fi` to denote the command `if` e `then` C `else skip fi`.

We now move to instantiating the other components of our program model for a given command. Commands themselves represent control locations in our instantiation, yet as our program model utilises the initial control location to model how critical information enters an execution through the high variables, we introduce for any command C an additional fresh control location $\mathfrak{st}_C$ and extend our above defined semantic function to transfer from $\mathfrak{st}_C$ with any state to C without changing the state ($\forall \sigma \in \Sigma \colon [\![\mathfrak{st}_C, \sigma]\!] = (C, \sigma)$). In order to avoid cluttering the program model for a command with infinitely many unreachable control locations, we do not utilise the full set of commands plus fresh start locations as control locations for the program model of a command but first define a global control flow abstraction on the set of commands as a syntactic approximation of the semantics and then utilise the set of reachable commands in this approximation as the control locations and the global control abstraction restricted to those as control abstraction for the instantiation.

Definition 4.10. The global control flow abstraction on all commands $\rightarrow \subseteq \mathcal{C} \times \mathcal{C}$ is inductively defined by:

$$\texttt{skip} \rightarrow \texttt{skip} \qquad v = e \rightarrow \texttt{skip}$$

$$\texttt{print}\ e \rightarrow \texttt{skip}$$

$$\texttt{if}\ e\ \texttt{then}\ C_t\ \texttt{else}\ C_f\ \texttt{fi} \rightarrow C_t \qquad \texttt{if}\ e\ \texttt{then}\ C_t\ \texttt{else}\ C_f\ \texttt{fi} \rightarrow C_f$$

$$\texttt{while}\ e\ \texttt{do}\ C\ \texttt{od} \rightarrow \texttt{skip} \qquad \texttt{while}\ e\ \texttt{do}\ C\ \texttt{od} \rightarrow C; \texttt{while}\ e\ \texttt{do}\ C\ \texttt{od}$$

$$\begin{aligned} C_1; C_2 &\rightarrow C_2 && \text{if } C_1 \rightarrow \texttt{skip} \\ C_1; C_2 &\rightarrow C_1'; C_2 && \text{if } C_1 \rightarrow C_1' \wedge C_1' \neq \texttt{skip} \end{aligned}$$

Based on this, the set of control locations for a command $C \in \mathcal{C}$ is defined as

$$N_C = \{C' \in \mathcal{C} \mid C \rightarrow^* C'\} \cup \{\mathfrak{st}_C\},$$

where $\mathfrak{st}_C$ is the fresh control location fixed above, from which the semantic moves to C. The control flow abstraction E_C is defined as

$$E_C = \{(\mathfrak{st}_C, C)\} \cup \{(C_1, C_2) \in N_C \mid C_1 \rightarrow C_2\}.$$

In order to define the data abstraction (def, use) we first fix the set of *low* variables $L \subseteq \mathrm{Var}$ and let H be $\mathrm{Var} \setminus L$ as Assumption 7 of our program model requires that these are exactly the variables read and written at the initial control location. For commands themselves the variables read or written only depend upon the head of the command, which is the first command in a sequential composition of commands or the command itself for basic commands. All variables occurring in an expression of the head make up the variables read by a command. Only commands whose head is an assignment write any variables, which in that case is the single variable occurring on the left-hand side of the assignment.

Definition 4.11 (Data abstraction)**.** The set of variables appearing in an expression e is denoted by $\mathrm{fv}(e)$ and inductively defined through

$$\mathrm{fv}(e) = \begin{cases} \emptyset & \text{if } e = c \in \mathcal{CT}, \\ \{v\} & \text{if } e = v \in \mathrm{Var}, \\ \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) & \text{if } e = e_1 \mathit{~op~} e_2. \end{cases}$$

The head of a command C, denoted by $\mathrm{hd}(C)$, is defined as

$$\mathrm{hd}(C) = \begin{cases} C' & \text{if } C = C'; C'', \\ C & \text{otherwise.} \end{cases}$$

With these the set of variables read at a control location C is defined through

$$\mathrm{use}(C) = \begin{cases} H & \text{if } C = \mathfrak{st}_{C'}, \\ \emptyset & \text{if } \mathrm{hd}(C) = \texttt{skip}, \\ \mathrm{fv}(e) & \text{if } \mathrm{hd}(C) = \texttt{print}~e, \\ \mathrm{fv}(e) & \text{if } \mathrm{hd}(C) = (v = e), \\ \mathrm{fv}(e) & \text{if } \mathrm{hd}(C) = \texttt{if}~e~\texttt{then}~C_1~\texttt{else}~C_2~\texttt{fi}, \\ \mathrm{fv}(e) & \text{if } \mathrm{hd}(C) = \texttt{while}~e~\texttt{do}~C'~\texttt{od}. \end{cases}$$

And the set of variables written at a control location is defined by

$$\mathrm{def}(C) = \begin{cases} H & \text{if } C = \mathfrak{st}_{C'}, \\ \{v\} & \text{if } \mathrm{hd}(C) = (v = e), \\ \emptyset & \text{otherwise.} \end{cases}$$

As mentioned above, we utilise the occurring `print` statements to define the attacker model. At any command whose head is a `print` statement, the corresponding observation is defined by evaluating the expression from that statement against the reached state.

Definition 4.12 (Observation)**.** The *attacker model corresponding to a command C* is denoted as obs_C and defined by

$$\mathrm{obs}_C = \{(C', \sigma \mapsto \mathrm{eval}(e,\sigma)) \mid C' \in N_C \wedge \mathrm{hd}(C') = \texttt{print}\ e\}.$$

With this we have all components to define the program corresponding to a command.

Definition 4.13. Given a command $C \in \mathcal{C}$ and $L \subseteq \mathrm{Var}$ the corresponding program P_C is defined as $P_C = (\Sigma, N_C, E_C, \mathfrak{st}_C, \texttt{skip}, [\![.]\!], \mathrm{def}, \mathrm{use}, L, \mathrm{obs}_C)$.

Lemma 4.2. *The program corresponding to a command is admissible in the sense of Definition 4.2.*

Proofsketch. We have $\mathfrak{st}_C \in N_C$ by definition. By induction one obtains $C' \to^* \texttt{skip}$ for any $C' \in \mathcal{C}$ and thereby $\texttt{skip} \in N_C$ as well as $N_C \times \{\texttt{skip}\} \subseteq E_C^*$, as required by Assumption 2. Assumption 1, that the semantic maps the terminal control location ($\texttt{skip}$) to itself without changing the reached state, follows directly from Definition 4.9. Assumption 3, that all steps in the semantics are matched by E_C, follows by simple case distinction from its definition, as does Assumption 4, that `skip` only reaches itself in E_C. Assumption 5, that at any control location n, no variable besides those in $\mathrm{def}(n)$ are changed by the semantics, follows as the semantics only updates the left-hand side variable in assignments. As the semantics of expressions only depends upon the values of the occurring variables, Assumption 6 follows. Finally Assumption 7, that $\mathrm{def}(\mathfrak{st}_C) = \mathrm{use}(\mathfrak{st}_C) = \mathrm{Var} \setminus L$, trivially follows from Definition 4.11. □

To conclude this chapter we give a more involved example and introduce a graphical notation to depict the programs corresponding to commands in Example 4.2.

Example 4.2. For the command from Figure 4.1 we depict in Figure 4.2 a graphical notation that represents the control flow abstraction of the corresponding program. The nodes of the graph represent the control locations of the program and its edges are those defined by the control abstraction. We do not annotate the nodes with the complete command that corresponds to the control location but only with a part of the commands head that is sufficient to reconstruct the command. For that purpose the outgoing edges of `if` and `while` commands are labelled with `t` and `f` to distinguish the `then` and `else` branch. We do not formalise the procedure to reconstruct the command from this representation, which we only utilise in examples where it should be obvious, how the original command can be obtained.

```
if b
then
  i = 0;
  x = h
else
  i = 1
fi;
while i < 2*u
do
  i = i + 1;
  z = x;
  x = y;
  y = z
od;
print y
```

Figure 4.1: Command

$$\begin{tikzcd}
 & & \mathfrak{st} \arrow[d] & \\
 & & \texttt{if b} \arrow[dll, "\texttt{t}"'] \arrow[dr, "\texttt{f}"] & \\
\texttt{i = 0} \arrow[d] & & & \texttt{i = 1} \arrow[dl] \\
\texttt{x = h} \arrow[rr] & & \texttt{while i < 2*u} \arrow[dl, "\texttt{t}"'] \arrow[dr, "\texttt{f}"] & \\
 & \texttt{i = i + 1} \arrow[d] & & \texttt{print y} \arrow[d] \\
 & \texttt{z = x} \arrow[d] & & \texttt{skip} \arrow[loop right] \\
 & \texttt{x = y} \arrow[d] & & \\
 & \texttt{y = z} \arrow[uuuur, bend left] & &
\end{tikzcd}$$

Figure 4.2: Graph Notation

We also utilise the labels from the graph notation to refer to the nodes themselves. In order to obtain a program through Definition 4.13 we need to fix the set of low or respectively high variables, which we set to $H = \{h\}$. An exemplary execution of this program is depicted in Figure 4.3. The execution produces the observation $\text{obs}(\sigma) = 7$, which is the initial value of y. Note, that in this example the heads of the control locations and their labels are unique which must not be the case in all examples but the actual referred to control location will be clear from the context.

$$\begin{array}{llll}
(& \mathtt{st}, & \sigma &), \\
(& \mathbf{if}\ \mathtt{b}, & \sigma &), \\
(& \mathtt{i\ =\ 0}, & \sigma &), \\
(& \mathtt{x\ =\ h}, & \sigma\{0/i\} &), \\
(& \mathbf{while}\ \mathtt{i} < \mathtt{2{\star}u}, & \sigma\{0/i, 3/x\} &), \\
(& \mathtt{i\ =\ i\ +\ 1}, & \sigma\{0/i, 3/x\} &), \\
(& \mathtt{z\ =\ x}, & \sigma\{1/i, 3/x\} &), \\
(& \mathtt{x\ =\ y}, & \sigma\{1/i, 3/x, 3/z\} &), \\
(& \mathtt{y\ =\ z}, & \sigma\{1/i, 7/x, 3/z\} &), \\
(& \mathbf{while}\ \mathtt{i} < \mathtt{2{\star}u}, & \sigma\{1/i, 7/x, 3/z, 3/y\} &), \\
(& \mathtt{i\ =\ i\ +\ 1}, & \sigma\{1/i, 7/x, 3/z, 3/y\} &), \\
(& \mathtt{z\ =\ x}, & \sigma\{2/i, 7/x, 3/z, 3/y\} &), \\
(& \mathtt{x\ =\ y}, & \sigma\{2/i, 3/x, 7/z, 3/y\} &), \\
(& \mathtt{y\ =\ z}, & \sigma\{2/i, 3/x, 7/z, 7/y\} &), \\
(& \mathbf{while}\ \mathtt{i} < \mathtt{2{\star}u}, & \sigma\{2/i, 3/x, 7/z, 7/y\} &), \\
(& \mathbf{print}\ \mathtt{y}, & \sigma\{2/i, 3/x, 7/z, 7/y\} &), \\
(& \mathbf{skip}, & \sigma\{2/i, 3/x, 7/z, 7/y\} &)^{\omega}.
\end{array}$$

Figure 4.3: Execution for σ with $\sigma(b) = 1$, $\sigma(h) = 3$, $\sigma(y) = 7$ and $\sigma(u) = 1$.

One can observe that the program actually is secure in the sense of Definition 4.5. While the initial value of h is first written to x and from there written to y — a variable which can be observed by the attacker eventually — the values of x and y are swapped in each iteration of the loop, which is executed an even number of times whenever h was written to x. The attacker therefore either observes the initial value of y, whenever the initial value of b is non-zero, or the initial value of x otherwise. We will illustrate with Example 7.7 in Chapter 7 how our approach leads to an analysis capable of certifying this when instantiating it with a suitable safety analysis that is able to reason about evenness.

5 Characterisation of Information Flows

In this chapter we develop our central characterisation of information flows in programs that provides the basis of our approach. For an admissible program in the sense of Definition 4.2 we describe how variations in initial states can be propagated within executions to produce distinguishable observations that are prohibited by our definition of security.

Our approach is inspired by program dependency graphs and also uses data and control dependencies based on abstractions of control flow and data, which we already fixed with the program model, to track how information is propagated by a program. In contrast to the approach used in program dependency graphs, which syntactically describe dependencies between control locations, our approach instead defines dependencies within executions themselves where semantic properties can still be fully exploited. The latter is also possible because our characterisation directly considers pairs of executions that might violate the security property, and for these it defines critical points in the executions that are influenced by confidential information and between which this information is propagated along dependencies postulated by the abstractions of control flow and data but is precise in the sense that it only requires propagation along those dependencies where there is actually semantically observable difference between the executions. This yields a characterisation of critical executions, which then can be abstracted in various ways to function as a heuristic to obtain sound and effective analyses for security, which are free to exploit arbitrarily precise semantic properties while avoiding the danger of running into soundness problems as mentioned earlier in Chapter 2 for the combination of path conditions with precise program analysis techniques.

We now move towards defining our characterisation of critical executions. To this end we fix throughout this chapter a program $(\Sigma, N, E, \mathfrak{st}, \mathfrak{te}, [\![.]\!], \mathrm{def}, \mathrm{use}, L, \mathrm{obs})$, admissible in the sense of Definition 4.2.

As mentioned we intend to define control and data dependencies within executions to track information flows based on the abstractions of control flow and data. Within single executions, where there is no additional structure to the control locations, it is hard to define what control is supposed to mean. In our setting this additional structure is provided by the control flow abstraction E together with the terminal control location $\mathfrak{tc}$, which we already included in the definition of admissible programs. The control flow abstraction allows us to define control dependence through the use of post dominance. The idea is that if a control location is post dominated by another one, the former has no direct influence on whether the later will be executed or not and therefore the absence of post dominance between control locations in an execution can be used to identify control dependence. We utilise the strict post dominance relation, which is the irreflexive version of the post dominance relation. A control location $n \in N$ strictly post dominates another control location $m \in N$, written $n \xrightarrow{pd} m$, if n and m are distinct and n appears on any path from m to the terminal control location $\mathfrak{tc}$.

Definition 5.1 (Strict post dominance)**.** The strict post dominance relation $\xrightarrow{pd}$ on N induced by $(N, E, \mathfrak{tc})$ is defined through

$$n \xrightarrow{pd} m \Leftrightarrow n \neq m \wedge \forall (n_i)_{i \geq 0} \in \Pi, k \in \mathbb{N} \colon n_0 = m \wedge n_k = \mathfrak{tc} \Rightarrow \exists l \leq k \colon n_l = n,$$

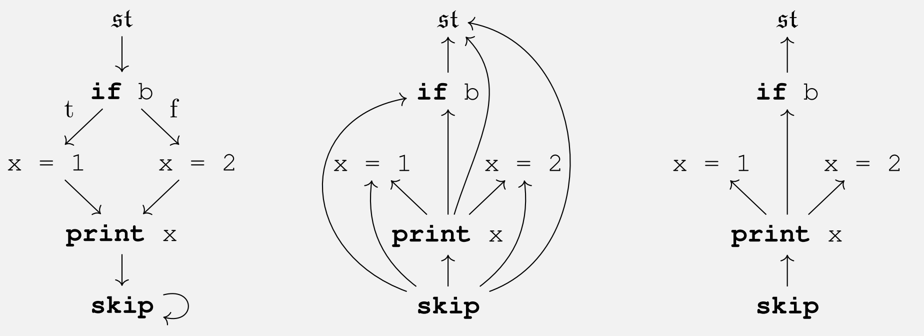
where $\Pi = \{(n_i)_{i \geq 0} \in N^\omega \mid \forall i \colon (n_i, n_{i+1}) \in E\}$ is the set of infinite paths in (N, E).

Note that as all control locations reach the looping terminal control location in (N, E), any finite path can be extended to a corresponding infinite path, such that the definition is equivalent to a version based on finite paths. We utilise infinite paths here as we will later only consider infinite paths. Also note that the strict post dominance relation can be succinctly represented as the transitive closure of the post dominance tree and can be computed in quasi linear time[1] using the algorithm by Lengauer and Tarjan [31] if N is finite. The post dominance relation and tree for a simple program from our command language are illustrated in Example 5.1.

Example 5.1. Figure 5.1 depicts a program (a), which corresponds to the command **if** `b` **then** `x = 1` **else** `x = 2` **fi; print** `x`, together with its strict post dominance relation (b) as well as its post dominance tree (c). Note how

[1]That is in time $\mathcal{O}(|E| \cdot a(|E|, |N|))$ where a is an inverse of Ackermann's function.

the two assignments to x, which lie in the conditional branches, are the only control locations not post dominating their predecessors and how the post dominance relation is the transitive closure of the post dominance tree.



(a) Program (b) Post dominance relation (c) Post dominance tree

Figure 5.1: Program with post dominance relation and tree.

Based on the definition of post dominance between control locations, we now move to define control dependencies within executions. As our program model utilises a deterministic semantics, any *point* in any execution can be uniquely identified through the initial state and the number of steps taken from there. Recall that to this end we already introduced in Chapter 4 the convention that whenever a context fixes an initial state σ, we fix $(n_i)_{i\geq 0}$ and $(\sigma_i)_{i\geq 0}$ such that $((n_i,\sigma_i))_{i\geq 0} = (\llbracket \mathfrak{st}, \sigma \rrbracket^i)_{i\geq 0}$. A control dependency is identified by a triple (σ, i, j) consisting of the initial state σ and two indices i and j where i is a step in the execution of σ where it was required — according to the control flow abstraction E — to take the branch from control location n_i to n_{i+1} in order to reach the control location n_j in step j. The latter is the case if n_j lies on a branch that starts with n_{i+1} but there exists another branch of n_i that does not contain n_j and it is also the case that the location where the branches of n_i merge has not been reached before step j. This is expressed through the requirement that there appears no strict post dominator of n_i between the steps i and j. We denote this with $i \xrightarrow{cd}_\sigma j$, which is formally defined in the following definition.

Definition 5.2 (Control dependency)**.** For $i, j \in \mathbb{N}$ we define that *i is a control dependency of j in the execution corresponding to* $\sigma \in \Sigma$, denoted by $i \xrightarrow{cd}_\sigma j$, through

$$i \xrightarrow{cd}_\sigma j \Leftrightarrow i < j \wedge \nexists k \in [i, j]\colon n_k \xrightarrow{pd} n_i.$$

Note that this definition closely corresponds to the definition of (transitive) control dependencies between control locations in program dependence graphs. This connection is formalised by Lemma 7.1 in Section 7.1 and discussed there. Similar notions called dynamic control dependence for paths within the control flow graph have for example been used by Xin and Zhang [51] for the purpose of dynamic program slicing [1] as well as in the soundness proofs for PDG-based information flow control by Wasserrab et al. [49] and Giffhorn and Snelting [20].

A potentially surprising property of this definition of control dependence in executions is the fact that the instances of the terminal control location are control dependencies of all their successors as we observe in Example 5.2. While these control dependencies on the instances of the terminal control location might not seem canonical, as there is no branching happening after all, they emerge naturally from the definition through post dominance, which was chosen this way deliberately, as it allows us to handle terminating and non-terminating executions in the same way in several theorems down the line. Beside its surprising nature it has no downsides to the remaining development.

Example 5.2. Again consider the program from Figure 5.1a. Let σ be an initial state with $\sigma(b) = 1$. This gives rise to an execution $((n_i, \sigma_i))_{0 \le i}$ whose first steps are depicted in Figure 5.2 together with the control dependencies resulting from the post dominance relation from Example 5.1. Until step 4 the only control location that does not post dominate its predecessor is n_2, wherefore we have $1 \xrightarrow{cd}_\sigma 2$. For all $i \ge 4$ we have that $n_i =$ **skip** and as the strict post dominator relation is irreflexive, we have $i \xrightarrow{cd}_\sigma j$ for $4 \le i < j$.

$$\begin{tikzcd}
i & 0 & 1 \arrow[r, "cd", "\sigma"'] & 2 & 3 & 4 \arrow[r, "cd", "\sigma"'] \arrow[rr, bend left, "cd"] & 5 \arrow[r, "cd", "\sigma"'] & \dots \\
n_i & \texttt{st} & \textbf{if}\ \texttt{b} & \texttt{x = 1} & \textbf{print}\ \texttt{x} & \textbf{skip} & \textbf{skip} & \dots \\
\sigma_i & \sigma & \sigma & \sigma & \sigma\{1/x\} & \sigma\{1/x\} & \sigma\{1/x\} & \dots
\end{tikzcd}$$

Figure 5.2: Execution with control dependencies.

We now move to defining the second kind of dependence which we utilise for our definition of critical executions, data dependencies. While control dependencies model an, as it is sometimes referred to, *implicit flow* of information within a program, where discernible states between two executions differ because the executions took different paths to reach them, data dependencies describe how differences in states are *directly* propagated through the semantic state transformers that are associated with the control locations, as they can propagate information about the values of variables that are read at a control location to the values of the variables which are written there. In the same way that the definition of control dependence was based on the control flow abstraction, the definition of data dependence, will be based on the data abstractions (def and use), which we fixed in Definition 4.2.

As we did not motivate the assumptions made for the data abstraction in Definition 4.2 before, we do so here. Similarly to how we used an indistinguishability relation on the initial states in the definition of security, which was defined through fixing the set of low variables L, we utilise the sets of variables provided by the data abstraction to do so within executions, that is based on the sets of variables read, respectively written, at the reached control locations. Assumptions 5 to 7 of Definition 4.2 then state how indistinguishability under these relations is preserved by the semantics as well as how indistinguishability locally leads to the same behaviour. Assumption 5 requires that the set of variables written at a control location ($\text{def}(n)$), encompasses all variables whose values can be changed by the associated state transformer, which guarantees that based on all other variables, input and output state are indistinguishable. Assumption 6 then requires that indistinguishability between input states, based on the variables read at a control location, must be sufficient to also ensure indistinguishability between the output states, based on the variables written at the control location. Assumption 6 moreover requires that under these circumstances, the semantics continues to the same control location, as well as that any possible observations produced must be identical. Finally Assumption 7 makes the link on the other end to the indistinguishability relation on the initial states, in that the variables read and written at the initial control location, are exactly the non-low variables.

For programs from our command language we saw in Lemma 4.2 that these assumptions are fulfilled by setting the set of variables written by a command to the set containing only the variable appearing on the left-hand side if the head of the command is an assignment and the read variables to the variables appearing in the contained expression if the head of the command is an assignment, branch or print command, as done in Definition 4.11.

The data dependency relation tracks the direct flow of information from a point where a variable is written to points where it is read, a setting in which distinguishability based on the variables read at a point can be propagated to another. As in our definition of control dependencies, we identify the execution through the initial state and besides the two indices forming the dependency, we additionally include the variable through which the dependency is created, in order to enable us to impose additional constraints on it, when using the dependency in later definitions. A data dependency is therefore a four-tuple (σ, i, j, v) where σ is the initial state, i is smaller than j and the variable v is written by n_i, read by n_j but not redefined by any n_k for k between i and j. We denote this by $i \xrightarrow{dd_v}_\sigma j$, which is formally defined in the following definition.

Definition 5.3 (Data dependency)**.** For $i, j \in \mathbb{N}$ we define that i *is a data dependency of* j *via variable* $v \in \mathrm{Var}$ *in the execution corresponding to* $\sigma \in \Sigma$, denoted by $i \xrightarrow{dd_v}_\sigma j$, through

$$i \xrightarrow{dd_v}_\sigma j \Leftrightarrow i < j \wedge v \in \mathrm{def}(n_i) \cap \mathrm{use}(n_j) \wedge \forall k \in (i, j) \colon v \notin \mathrm{def}(n_k)$$

We illustrate this definition in the following example.

Example 5.3. Again consider the program from Figure 5.1a while assuming that $H = \{b\}$. Definition 4.11 then yields that the data abstraction has the following form. We have $\mathrm{use}(\mathfrak{st}) = \mathrm{def}(\mathfrak{st}) = \mathrm{use}($`if b`$) = \{b\}$ as well as that $\mathrm{def}($`x = 1`$) = \mathrm{def}($`x = 2`$) = \mathrm{use}($`print x`$) = \{x\}$ and all other control locations are mapped to the empty set by both maps.

In Figure 5.3 the same execution as in Example 5.2 is depicted together with the data abstraction of the reached control locations and the resulting data dependencies. We have only two data dependencies. Firstly we have $0 \xrightarrow{dd_b}_\sigma 1$ as due to $H = \{b\}$ we have $b \in \mathrm{def}(\mathfrak{st}) \cap \mathrm{use}($`if b`$)$ and secondly $2 \xrightarrow{dd_x}_\sigma 3$ as $x \in \mathrm{def}($`x = 1`$) \cap \mathrm{use}($`print x`$)$.

i	$0 \xrightarrow[\sigma]{dd_b}$	1	$2 \xrightarrow[\sigma]{dd_x}$	3	4	5	$\dots$
n_i	$\mathfrak{st}$	`if b`	`x = 1`	`print x`	`skip`	`skip`	$\dots$
$\mathrm{def}(n_i)$	$\{b\}$	$\emptyset$	$\{x\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\dots$
$\mathrm{use}(n_i)$	$\{b\}$	$\{b\}$	$\emptyset$	$\{x\}$	$\emptyset$	$\emptyset$	$\dots$
σ_i	σ	σ	σ	$\sigma\{1/x\}$	$\sigma\{1/x\}$	$\sigma\{1/x\}$	$\dots$

Figure 5.3: Execution with data dependencies.

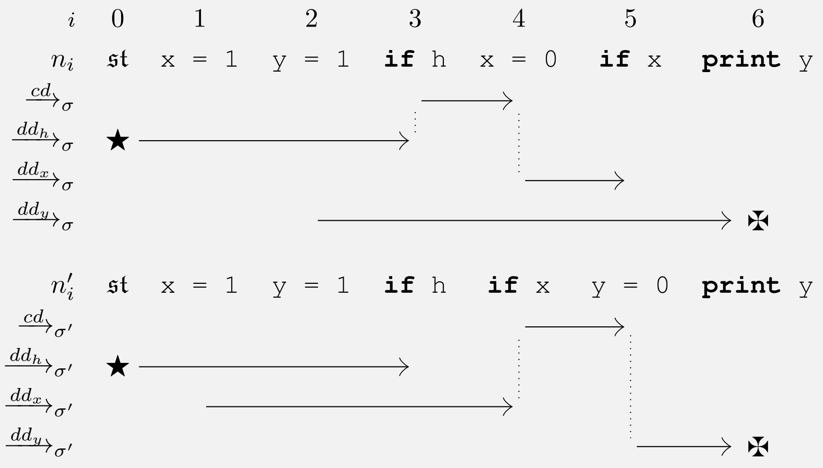
In the case of program dependence graphs, which motivated our approach, in order to guarantee the security of a program, it suffices to track data and control dependencies between control locations and verify that one cannot construct a path from a source reading high data (which in our case is the initial control location in the first step) to a sink producing an observation (in our case an observable control location). Unfortunately, the corresponding property does not hold for data and control dependencies within executions. While in our running example, where the program is insecure as it outputs 2 if the initial value of b is 0 and 1 otherwise, we actually have a path from the initial step to one reaching an observable control location with the dependencies described in Examples 5.2 and 5.3, namely $0 \xrightarrow{dd_b}_\sigma 1 \xrightarrow{cd}_\sigma 2 \xrightarrow{dd_x}_\sigma 3$ with $n_3 \in \mathrm{dom}(\mathrm{obs})$, this is not a sound criterion in the general case as we observe in Example 5.4.

Example 5.4. Consider the program in Figure 5.4 during whose execution an attacker can observe whether the initial value of h was 0 or not.

```
x = 1; y = 1;
if h then x = 0 fi;
if x then y = 0 fi;
print y
```

Figure 5.4: Insecure program for $H = \{h\}$.

The program is not secure for $H = \{h\}$, as we have for σ with $\sigma(h) = 1$ and $\sigma' = \sigma\{0/h\}$ that $\sigma =_L \sigma'$ but $\mathrm{obs}(\sigma) = 1 \neq 0 = \mathrm{obs}(\sigma')$. Figure 5.5 shows the data and control dependencies in the first steps of the executions from σ and σ'. Inspecting these dependencies we observe that in neither execution the observable **print** y executed in step number 6 (✠) can be reached through our dependencies from the initial step (★), which is the source for high information. The problem arises, as in the first execution we miss the fact that the value of x, at the execution of **if** x in step number 5, influences the value of y, at the execution of **print** y in step number 6, and also due to the fact that in the second execution the dependencies do not depict that the value of h, at the execution of **if** h in step number 3, influences the value of x, at the execution of **if** x in step number 4. In both cases at the latter step, the value of the variable reflects that it has not been updated and thereby propagates information about the value of the variable governing the branch.



i 0 1 2 3 4 5 6
n_i st x = 1 y = 1 if h x = 0 if x print y
cd σ
dd_h σ
dd_x σ
dd_y σ
n'_i st x = 1 y = 1 if h if x y = 0 print y
cd σ'
dd_h σ'
dd_x σ'
dd_y σ'

Figure 5.5: Dependencies in the executions of σ and σ'.

In Example 5.4 we observe how information can be propagated within an execution by not taking a branch that might update a variable. Moreover we see, that if this happens in two executions which violate our security property, the afore-defined dependencies themselves, do not directly suffice to track this propagation within any single execution. We notice however, that in these cases there always exists a control dependency followed by a data dependency in the other execution such that the source of the control dependency and the sink of the data dependency appear in both executions. This also holds in general as we will ultimately prove. Our approach therefore is to track these dependencies between matching steps, for a suitable definition of matching steps, in two executions simultaneously, which we prove to be a precise semantic property in the sense that it captures exactly those matching steps where different data is read. In a subsequent step we then approximate this characterisation to obtain more effective criteria based on single executions for the implementation of practical analyses.

A challenge that arises with this approach of considering dependencies in two executions simultaneously is the matching of steps between executions. In our example matching steps was rather simple as there were no loops and therefore any control location other than the terminal control location could only appear once in each execution. In the general case loops and branching make this matching of steps between arbitrary executions less trivial because of shifted and duplicated instructions. We utilise what Giffhorn [19] already used for this purpose, the control slice. The control slice of an index within an execution consists of the reached control location plus the sequence of control locations reached by indices upon which the index is control dependent. We then consider indices in two executions matching, if they have the same control slice.

Definition 5.4 (Control slice)**.** The *control slice of an index i within an execution corresponding to $\sigma \in \Sigma$* is the sequence $\mathrm{cs}_i^\sigma = (n_{i_j})_{j \leq k}$ of control locations, where $(i_j)_{j \leq k}$ is the maximal strictly ascending sequence of indices such that $i_k = i$ and $\forall j < k\colon i_j \xrightarrow{cd}_\sigma i$. Based on this the *control slice equivalence* between indices of two executions is defined as $i =^{\mathrm{cs}}_{\sigma,\sigma'} i'$ if and only if $\mathrm{cs}_i^\sigma = \mathrm{cs}_{i'}^{\sigma'}$ and in this case we say that i in execution of σ is *matched* by i' in the execution of σ'.

The control slice is useful to identify instances of control locations within different executions for the following three reasons. Firstly, it is injective on every execution (Lemma 6.15). Secondly, it determines the order in which instances of control locations are visited by executions (Lemma 6.16). Thirdly, it has the — for information flow purposes — useful property that if for an index there does not exist an index with a matching control slice in another execution then there must either be a non-terminating loop or some control dependency of the unmatched index took a different branch in the other execution (Lemma 6.28). This will allow us to follow the propagation of information in two executions in a lockstep fashion by always stepping from one pair of matched indices to the next while tracking data and control dependencies.

We now have all tools assembled to define our criterion for critical executions or more precisely the critical points within these. At this point we strive to make our criterion as precise as possible, which is aided by the fact that we still work semantically. This will allow later analyses to exploit any kind of additional semantic information to increase its precision without having to worry about soundness problems as the underlying characterisation did not rely on any imprecision to obtain its soundness. As mentioned, we will consider two executions simultaneously and only advance from one pair of matching indices to the next while tracking data and control dependencies. To achieve maximal precision we require that at each such pair there is an actual diverging value for a variable read at that point, that is the reached states are distinguishable based on the variables read at the reached control location, which must be the same in both executions because the indices match.

Our inductive definition begins at the first step in the executions for two low equivalent but distinct input states. As all executions start at the same control location $\mathfrak{st}$ and because the first step in an execution cannot have any control dependencies, the control slices of the first steps between any two executions match and as our data abstraction by assumption defines the set of variables read by the initial control location to be exactly the high variables, the states are distinguishable based on the variables read. In the inductive step we consider the two cases we observed in our examples. Either both executions propagate the diverging value directly via corresponding data dependencies on the same variable to another pair of matching indices or the executions diverge by taking different branches and in at least one of the executions a control dependent index is reached that is a data dependency of a matched index, which we then call a data control dependency on the matching indices.

Definition 5.5 (Critical executions/critical matching indices)**.** The data dependency relation is lifted to pairs of matching indices $i =^{\text{cs}}_{\sigma,\sigma'} i'$ and $j =^{\text{cs}}_{\sigma,\sigma'} j'$, parameterised by initial states $\sigma, \sigma' \in \Sigma$ and $v \in \text{Var}$ via

$$(i,i') \xrightarrow{dd_v}_{\sigma,\sigma'} (j,j') \iff i \xrightarrow{dd_v} j \wedge i' \xrightarrow{dd_v} j'. \tag{5.1}$$

Moreover, the *data control dependency relation* on matching indices is defined by

$$(i,i') \xrightarrow{dcd_v}_{\sigma,\sigma'} (j,j') \Leftrightarrow \begin{array}{c} n_{i+1} \neq n'_{i'+1} \wedge \exists k\colon i \xrightarrow{cd}_{\sigma} k \xrightarrow{dd_v}_{\sigma} j \wedge \\ \forall l,l',k'\colon i < l =^{\text{cs}}_{\sigma,\sigma'} l' \leq k' < j' \Rightarrow v \notin \text{def}(n'_{k'}). \end{array} \tag{5.2}$$

With these, the *critical matching indices* $\bowtie_{\sigma,\sigma'} \subseteq \mathbb{N} \times \mathbb{N}$ for $\sigma, \sigma' \in \Sigma$ are inductively defined through:

$$\sigma =_L \sigma' \wedge \sigma \neq_H \sigma' \implies 0 \bowtie_{\sigma,\sigma'} 0 \tag{5.3}$$

$$\begin{pmatrix} i \bowtie_{\sigma,\sigma'} i' \wedge \\ j =^{\text{cs}}_{\sigma,\sigma'} j' \wedge \\ \sigma_j(v) \neq \sigma'_{j'}(v) \end{pmatrix} \wedge \begin{pmatrix} (i,i') \xrightarrow{dd_v}_{\sigma,\sigma'} (j,j') \vee \\ (i,i') \xrightarrow{dcd_v}_{\sigma,\sigma'} (j,j') \vee \\ (i',i) \xrightarrow{dcd_v}_{\sigma',\sigma} (j',j) \end{pmatrix} \Rightarrow j \bowtie_{\sigma,\sigma'} j' \tag{5.4}$$

We will step through this definition in detail as it is central to our development. The definition of data dependencies on matching indices in (5.1) is straightforward. It is meant to capture the case where different values are propagated by storing them in a variable that is read later. To this end both executions must have reached matching indices (that is the same location with the same control slice) and both must exhibit a data dependency on the same variable to another pair of matching indices. Note that while the executions reach matching indices at the beginning and end of the dependency, they do not need to progress in the same manner in-between.

The definition of data control dependencies on matching pairs in (5.2) is a little more involved. It captures the case where control flow differs and during this time one execution updates a variable that is later read at a point, where the control flow has merged again. To this end the definition in (5.2) requires that $n_{i+1} \neq n'_{i'+1}$, which ensures that the control flow actually differs and furthermore requires the existence of another index k, which is control dependent upon the matched index i and updates the variable v that is read at the matched index j. The definition also requires that there is no update to the variable v in the other execution after the control flow merged again with the first execution. This requirement is added because we strive to make the property as strict as possible and the existence of such an index would lead to another intermediate dependency to a pair of matching indices that lies before that update. Including this requirement provides us with an additional constraint on the execution that does not update the variable that can be exploited to optimise later approximations like the single execution property we derive. Note that in the case where the control flow is well-structured — in the sense that if a branch point has multiple branches, no two of them merge early that is all branches merge at a common post dominator of the branch point, which is always the case for programs derived from our command language as it only supports binary branching — the requirement that k' lies after a pair of matching indices is equivalent to k' not being control dependent upon i'.

The critical matching indices for initial states that coincide on the low variables L but differ on the high variables H are then defined inductively beginning at the matching indices 0 and 0. The inductive case then steps from two critical matching indices $i \bowtie_{\sigma,\sigma'} i'$ to two matching indices $j =^{\text{cs}}_{\sigma,\sigma'} j'$ whose corresponding reached states σ_j and $\sigma'_{j'}$ disagree on the value of some variable v on which there either is a data dependency or a data control dependency between the pairs of matching indices. Note that there is only one case for data dependencies as their definition is symmetric. Also note that in either case of (5.4) it holds that v is read at the location reached by j and j', as at least one of them is a target of a data dependency on v, which directly yields that the executions reached matching indices where reached states are distinguishable based on the variables read. We illustrate this definition in Example 5.5.

Example 5.5. We again consider the program and executions from Example 5.4. Figure 5.5 depicts the dependencies on indices, the resulting dependencies on matching indices and their criticality, which are derived as follows:

- We have $\sigma =_L \sigma'$ and $\sigma \neq_H \sigma'$ whereby with (5.3) it holds that $0 \bowtie_{\sigma,\sigma'} 0$.
- In both executions we have a data dependency of 3 on h upon 0 and as in both executions 3 has no control dependencies and reaches the same control location we have $3 =^{\text{cs}}_{\sigma,\sigma'} 3$ whereby it follows with (5.1) that $(0,0) \xrightarrow{dd_h}_{\sigma,\sigma'} (3,3)$ and as $\sigma_3(h) = 1 \neq 0 = \sigma'_3(h)$ this yields with (5.4) that $3 \bowtie_{\sigma,\sigma'} 3$.
- As for the pair $(3,3)$ it holds for the pair $(5,4)$ that $5 =^{\text{cs}}_{\sigma,\sigma'} 4$. It also holds that $n_4 = (\texttt{x = 0}) \neq (\textbf{\texttt{if}}\ \texttt{x}) = n'_4$ and with $3 \xrightarrow{cd}_\sigma 4 \xrightarrow{dd_x}_\sigma 5$ and there being no write of x in the execution of σ' between 3 and 4 it follows with (5.2) that $(3,3) \xrightarrow{dcd_v}_{\sigma,\sigma'} (5,4)$ whereby with $\sigma_5(x) = 1 \neq 0 = \sigma'_4(x)$ and (5.4) it follows that $5 \bowtie_{\sigma,\sigma'} 4$.
- Analogously we have $6 =^{\text{cs}}_{\sigma,\sigma'} 6$, $4 \xrightarrow{cd}_{\sigma'} 5 \xrightarrow{dd_y}_{\sigma'} 6$ and there is no write of y between 5 and 6 in the execution of σ, wherefore with (5.2) it holds that $(4,5) \xrightarrow{dcd_y}_{\sigma',\sigma} (6,6)$ (note the swapped roles of σ and σ') and therefore with $\sigma_6(y) = 1 \neq 0 = \sigma'_6(y)$ and (5.4), we have $6 \bowtie_{\sigma,\sigma'} 6$.

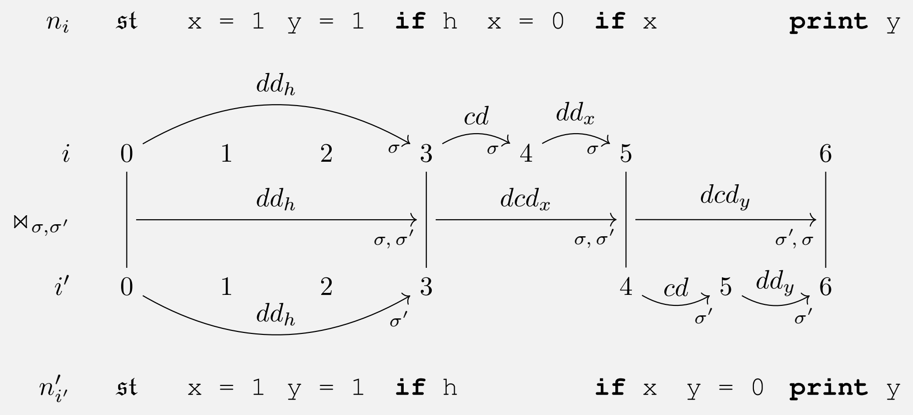


Figure 5.6: Propagation of $\bowtie_{\sigma,\sigma'}$.

Our definition has the desired property that it exactly captures the matching indices where the reached states are distinguishable based on the variables read in executions starting from low equivalent input states, which is captured in the following theorem.

Theorem 5.1.

$$\sigma =_L \sigma' \wedge i =^{\mathrm{cs}}_{\sigma,\sigma'} i' \wedge \sigma_i \neq_{\mathrm{use}(n_i)} \sigma'_{i'} \Leftrightarrow i \bowtie_{\sigma,\sigma'} i'$$

We dedicate the next chapter to the proofs of the central theorems from this chapter and therefore skip over them here. The proof for the above theorem can be found on Page 69 in the form of Corollary 6.34.

While our definition of critical matching indices captures the points in executions to which differences in the high input data are propagated, what we are actually interested in for our security property, is the question whether these differences lead to different observations for an attacker. The attacker is defined through a set of control locations at which, when they are reached during an execution, an observation is produced, which is determined by the values of the variables read at that location. Having characterised the points during an execution that are influenced by the high data, we can characterise which executions might finally lead to different observations for the attacker. We distinguish two cases based on whether the different observations are produced at matching points or not, both of which we will consider critical. That is either the executions reach matching points where the same control location reads different data that causes the executions to produce different observations or the executions produce the different observations at two points that do not match. In the latter case there must be some unmatched index in one of the executions, as matching indices always have to appear in the same order, wherefore the unmatched index must lie within a branch that was not taken by the other execution. We utilise this to define a notion of critical observable executions, which in general is somewhat stricter than our security property itself as the latter is chosen to be as simple as possible while the notion defined now is tailored to our structure driven approach. The definition of critical observable executions ignores the fact that our security property only considers the sequence of observations and that the same observations might be produced from different data at the same or at different locations. Our criterion instead requires that the observations are produced at the same points based on the same data. The critical observable executions, or more precisely the critical observable indices in executions, are defined as pairs of indices in two executions where the first index reached an observable control location and either the indices are critical matching indices themselves or the second index forms a critical matching pair with an index that is a control dependency of the observable index and after which the branch, inside which the observable index lies, was not taken by the other execution.

Definition 5.6 (Critical observable indices)**.** An index i in the execution corresponding to the initial state σ is called a *critical observable index* as witnessed by i' in σ', denoted $i \ltimes_{\sigma,\sigma'} i'$, if and only if it holds that

$$n_i \in dom(\text{obs}) \wedge \big(i \bowtie_{\sigma,\sigma'} i' \vee \big(\exists \iota\colon \iota \xrightarrow{cd}_{\sigma} i \wedge \iota \bowtie_{\sigma,\sigma'} i' \wedge n_{\iota+1} \neq n'_{i'+1}\big)\big)$$

and in this case the execution corresponding to σ is called a *critical observable execution.*

The non-existence of critical observable executions or equivalently the non-existence of critical observable indices, is the criterion targeted by our methodology. Example 5.6 illustrates in three cases how insecure programs exhibit critical observable executions.

Example 5.6. In the following examples we describe how one obtains critical observable indices for executions of some insecure programs described before.

(a) In Example 5.5 we observed that in the executions corresponding to σ and σ' the indices $(6,6)$ constitute a critical pair, that is $6 \bowtie_{\sigma,\sigma'} 6$. As n_6 is an observable control location, we obtain that 6 is a critical observable index in the execution corresponding to σ as witnessed by the index 6 in the execution corresponding to σ', that is $6 \ltimes_{\sigma,\sigma'} 6$.

(b) In Examples 5.2 and 5.3 we described some dependencies for the program `if b then x = 1 else x = 2 fi; print x` for σ with $\sigma(b) = 1$. Here we obtain with $\sigma' = \sigma\{0/b\}$ and $H = \{b\}$ that $(0,0) \xrightarrow{dd_b} (1,1)$ as well as $(1,1) \xrightarrow{dcd_x}_{\sigma,\sigma'} (3,3)$. It therefore holds that $3 \bowtie_{\sigma,\sigma'} 3$ and because $n_3 =$ `print x` $\in \text{dom(obs)}$, we obtain $3 \ltimes_{\sigma,\sigma'} 3$.
Note that $(1,1) \xrightarrow{dcd_x}_{\sigma',\sigma} (3,3)$ also holds (notice the swapped roles of σ and σ') wherefore we also have $3 \ltimes_{\sigma',\sigma} 3$. In cases like these our single execution approximation defined later will not depend upon a second execution.

(c) In order to illustrate the second case of Definition 5.6, where observations differ because an observation is missing from a terminating execution, consider the minimal program **if** h **then print** 1 **fi**. Assuming that $H = \{h\}$ and $\sigma(h) = 1$ as well as $\sigma' = \sigma\{0/h\} =_L \sigma$ it holds that $\text{obs}(\sigma) = 1 \not\lesssim \epsilon = \text{obs}(\sigma')$. It follows that the program is not secure as the execution for σ' terminates. We have $0 \bowtie_{\sigma,\sigma'} 0$ and $(0,0) \xrightarrow{dd_h}_{\sigma,\sigma'} (1,1)$ wherefore $1 \bowtie_{\sigma,\sigma'} 1$. With $n_2 =$ **print** $1 \neq$ `skip` $= n'_2$, $n_2 \in \text{dom}(\text{obs})$ and $1 \xrightarrow{cd}_\sigma 2$ we then obtain $2 \ltimes_{\sigma,\sigma'} 1$.

The following theorem states that the absence of critical observable executions is actually a sound criterion for the security of a program. More precisely it is the case that for any pair of executions which violate our security property at least one of them is a critical observable execution as witnessed by the other.

Theorem 5.2.

$$a)\quad \sigma =_L \sigma' \wedge \text{obs}(\sigma') \not\lesssim \text{obs}(\sigma) \wedge (\exists k\colon n_k = \mathfrak{te}) \Rightarrow \exists i, i'\colon i \ltimes_{\sigma,\sigma'} i' \vee i' \ltimes_{\sigma',\sigma} i$$

$$b)\quad \sigma =_L \sigma' \wedge \text{obs}(\sigma) \not\lesssim \text{obs}(\sigma') \wedge \text{obs}(\sigma') \not\lesssim \text{obs}(\sigma) \Rightarrow \exists i, i'\colon i \ltimes_{\sigma,\sigma'} i' \vee i' \ltimes_{\sigma',\sigma} i$$

The two parts of the above theorem capture the two cases of our security property for terminating and arbitrary executions (Definition 4.5) and directly imply the following corollary, which is the underlying correctness result for our approach. We defer the proof of the above theorem to the next chapter where it can be found in Corollaries 6.36 and 6.37.

Corollary 5.3 (Correctness). *If a program does not exhibit any critical observable executions, which is there are no i, i', σ, σ' fulfilling $i \ltimes_{\sigma,\sigma'} i'$, then the program is secure.* □

This result allows one to prove the security of a program by verifying that a program does not exhibit any critical observable executions. In Chapter 7 we will do this by applying abstractions to the definition of critical observable executions to obtain computable approximations. In Example 5.7 we apply the definitions directly in a semantic argument for the security of a program.

Example 5.7. We sketch how the definitions of critical and critical observable indices can be applied to verify the security of a program through the above correctness result in an ad hoc manner. Consider the program corresponding to the command from Figure 5.7 with $H = \{h\}$. We already considered a variant of this program in the introduction in Figure 1.1 as an example for a secure program where solely syntactic arguments are insufficient to verify its security.

```
if b then
    x = h; sec = 0
else
    x = y
fi;
if sec then
   print x
else
   print 0
fi
```

Figure 5.7: Secure program for $H = \{h\}$.

We have to assure ourselves that there can be no critical observable indices for this program. Starting with the inductive definition of critical pairs for arbitrary low equivalent but distinct σ and σ' it is the case that $0 \bowtie_{\sigma,\sigma'} 0$. From here, as $n_0 = \mathfrak{st}$ has only one successor which is a post dominator, there cannot exist any control dependencies. Due to $\text{def}(\mathfrak{st}) = \{h\}$, the only way to advance is via a data dependency on h, which is only read at `x = h`, a control location that can only be reached in the second step of an execution so that the data dependency in question would have to be $(0,0) \xrightarrow{dd_h}_{\sigma,\sigma'} (2,2)$. Again, as `x = h` has only one successor which is a post dominator and $\text{def}(n_2) = \{x\}$, the only way to advance is via another data dependency in both executions to **print** `x`, which would be reached by both executions in the fifth step. It can however not be the case that $(2,2) \xrightarrow{dd_x}_{\sigma,\sigma'} (5,5)$, as n_3 would be `sec = 0` wherefore $\sigma_4(sec) = 0$ and thereby $n_5 = [\![\textbf{if } \texttt{sec}, \sigma_4]\!]_l = \textbf{print } \texttt{0}$, which contradicts n_5 being **print** `x`.

The only critical pairs can be $0 \bowtie_{\sigma,\sigma'} 0$ where n_0 is $\mathfrak{st}$ and $2 \bowtie_{\sigma,\sigma'} 2$ where n_2 is `x = h`. As both are neither observable nor can have any control dependencies, there cannot be any critical observable indices and the program must be secure.

Note that in the above argument we did not rely on any properties between the two executions allowing it to be fully reduced to properties of single executions. In the following we will define single execution approximations that can handle this example as well.

Before we continue with the proofs of the theorems in the next chapter, we make a first step towards obtaining an effective analysis from our semantic characterisation. As we ultimately strive for decidable criteria for security, we are forced to abstract from our precise yet in general undecidable property. Here we perform a first step still within the semantics to move into this direction and in Chapter 7 we will present further abstractions that finally yield effective analyses. While our property from Definition 5.6 considers two executions simultaneously and defines a set of pairs of critical or unsafe executions, single execution safety properties, which only consider a single execution at a time and define a set of unsafe executions are often more tractable and enjoy better support from various program analysis techniques and frameworks. We therefore provide as a starting point for further abstractions an approximation of our two execution property by a single execution property. Alas, as we observed previously, it might not always be easily possible to identify a potential information flow within a single execution and as different abstractions might enable different possibilities to identify those, we still use a second execution locally to define an additional kind of dependency to capture these information flows.

We define the data control dependency relation on indices within a single execution analogously to the data control dependency relation on pairs of indices in Definition 5.5. They capture the case that in another execution there exist matching indices such that the former is a control dependency of an index that is a data dependency of the latter, which leads to different values being read at the latter index. As we consider single executions where we will also track data and control dependencies themselves instead of only tracking data and data control dependencies like we did for pairs of executions, we can make the definition of data control dependencies for single executions somewhat stronger than in the case of data control dependencies on pairs. Due to this the definition assumes that there is no update of the variable in the original execution at all while the definition on pairs of executions allowed updates to the variable after the control flow merged. As we did with data dependencies, which we partitioned over the variable used for the data dependency, we partition data control dependencies over the variable used and additionally over the alternative input state and the matching index in the witnessing execution in order to pose additional constraints on these. We then define a single execution approximation of our two execution characterisation of critical observable executions, which tracks data dependencies and control dependencies in single executions as well as data control dependencies to cover information flows not easily visible in an execution itself. The approximation still utilises the full semantics and initial states with execution indices to represent points within executions. We define two sets R and $C \subseteq \Sigma \times \mathbb{N}$, representing points in executions which might be reached by high information (R) as well as the observable subset of these (C).

Definition 5.7. For $i, j, i' \in \mathbb{N}$, $\sigma, \sigma' \in \Sigma$ and $v \in \mathrm{Var}$, the existence of a *data control dependency on v between i and j in the execution corresponding to σ witnessed by i' in σ'* is denoted as $i \xrightarrow{dcd_v^{\sigma',i'}}_\sigma j$ and defined by

$$\begin{aligned} i \xrightarrow{dcd_v^{\sigma',i'}}_\sigma j \iff & i =^{\mathrm{cs}}_{\sigma,\sigma'} i' \wedge \sigma =_L \sigma' \wedge (\forall k \in [i..j) \colon v \notin \mathrm{def}(n_k)) \wedge \\ & \exists k', j' \colon i' \xrightarrow{cd}_{\sigma'} k' \xrightarrow{dd_v}_{\sigma'} j' \wedge j =^{\mathrm{cs}}_{\sigma,\sigma'} j' \wedge \sigma_j(v) \neq \sigma'_{j'}(v). \end{aligned} \tag{5.5}$$

Using this relation, the reaching and critical reaching executions $R, C \subseteq \Sigma \times \mathbb{N}$ are defined inductively through:

$$(\sigma, 0) \in R \tag{5.6}$$

$$(\sigma, i) \in R \wedge i \xrightarrow{dd_v}_\sigma j \Rightarrow (\sigma, j) \in R \tag{5.7}$$

$$(\sigma, i) \in R \wedge i \xrightarrow{cd}_\sigma j \Rightarrow (\sigma, j) \in R \tag{5.8}$$

$$(\sigma, i) \in R \wedge (\sigma', i') \in R \wedge i \xrightarrow{dcd_v^{\sigma',i'}}_\sigma j \Rightarrow (\sigma, j) \in R \tag{5.9}$$

$$(\sigma, i) \in R \wedge n_i \in \mathrm{dom}(\mathrm{obs}) \Rightarrow (\sigma, i) \in C \tag{5.10}$$

This definition provides a safe approximation for critical observable executions as we record in the following theorem, whose proof we defer to the next chapter where it can be found as Corollary 6.39.

Theorem 5.4.

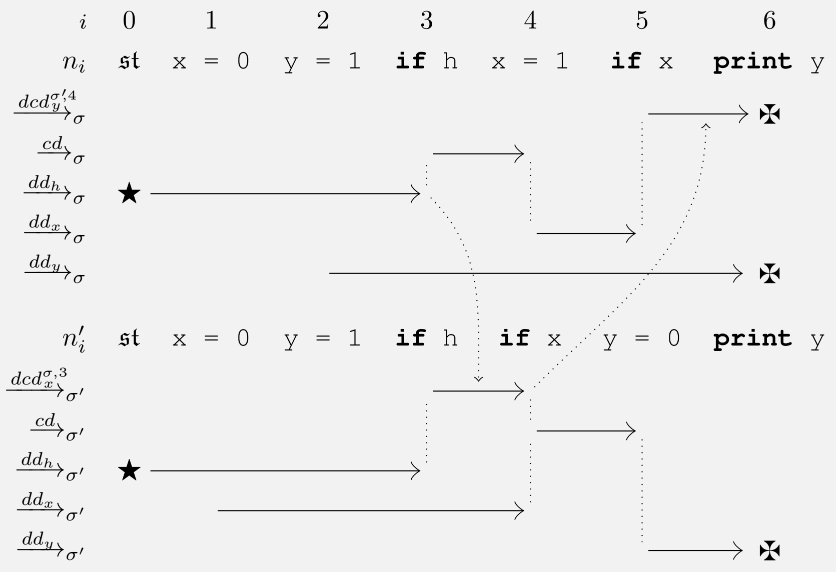
$$i \ltimes_{\sigma,\sigma'} i' \quad \Longrightarrow \quad (\sigma, i) \in C \wedge (\sigma', i') \in R$$

Combined with Corollary 5.3 this directly yields that emptiness of C is a sound criterion for security.

Corollary 5.5. *If C is empty, then the program is secure.* □

Applying stepwise abstractions to the definition of C in order to obtain computable criteria for security will be our focus in Chapter 7. We end this chapter with revisiting Example 5.4 using our newly defined single execution criterion.

Example 5.8. We again consider the program from Figure 5.4 with the same initial states σ and σ' as in Example 5.4. In Figure 5.8 we include, additionally to the data and control dependencies already listed in Example 5.4, the newly defined data control dependencies of the first steps of the executions of σ and σ'. With these additional dependencies we see in both executions that the initial read of the high data in step 0 (★) reaches the execution of the observable control location in step 6 (✠), whereby we obtain that both $(\sigma, 6)$ and $(\sigma', 6)$ lie in C and see that our criterion correctly deems the program insecure.



i 0 1 2 3 4 5 6
n_i st x = 0 y = 1 if h x = 1 if x print y
$\xrightarrow{dcd_y^{\sigma',4}}_\sigma$
$\xrightarrow{cd}_\sigma$
$\xrightarrow{dd_h}_\sigma$
$\xrightarrow{dd_x}_\sigma$
$\xrightarrow{dd_y}_\sigma$
n'_i st x = 0 y = 1 if h if x y = 0 print y
$\xrightarrow{dcd_x^{\sigma,3}}_{\sigma'}$
$\xrightarrow{cd}_{\sigma'}$
$\xrightarrow{dd_h}_{\sigma'}$
$\xrightarrow{dd_x}_{\sigma'}$
$\xrightarrow{dd_y}_{\sigma'}$

Figure 5.8: Dependencies in the executions of σ and σ'.

6 Correctness

In this chapter we provide proofs for the central Theorems 5.1 and 5.2 as well as for Theorem 5.4 from the previous chapter. The proofs presented in this chapter have also been formalised [39] and machine checked using the interactive theorem prover Isabelle/HOL.[1] We again assume the same context as before, that is we fix a program $(\Sigma, N, E, \mathfrak{st}, \mathfrak{te}, [\![.]\!], \text{def}, \text{use}, L, \text{obs})$, admissible in the sense of Definition 4.2.

We split the proof into three major parts. Firstly, in Section 6.1 we will study basic properties of paths, control dependence and the control slice to establish theorems to foster our understanding of how control dependencies determine the occurrence and order of operations in executions. In Section 6.2 we will then define the concept of contradicting executions to split the remaining proof into two parts, corresponding to the stepwise definition of critical observable indices which were based on critical matching pairs. Contradicting executions retain the idea of comparing differences in executions at steps with matching control slices but drop the tracking of data and control dependencies and thereby bridge the gap between the characterisation of critical matching pairs in Theorem 5.1 and the definition of critical observable indices. In order to exploit this we prove in Section 6.2 that insecure programs contain contradicting executions and then close the main proof in Section 6.3 by proving that contradicting executions themselves are critical observable executions. Finally, we quickly conclude in Section 6.4 with the correctness result for the single execution approximation by proving that critical observable executions are also considered critical by the single execution approximation.

[1] Note, that for historic reasons the Isabelle/HOL formalisation handles the terminal self loop in terminating executions in a slightly different way, which was, together with a few arguments, streamlined in the proofs presented here.

6.1 Properties of Paths

In this section we will establish required properties related to the control of a program. To this end we will study properties that are independent of the initial and reached states in the way that they only depend on the sequence of visited control locations. As the definitions of control dependence ($\xrightarrow{cd}$) (Definition 5.2), data dependence ($\xrightarrow{dd_v}$) (Definition 5.3) and the control slice (cs) (Definition 5.4) for initial states already only depend upon the sequence of control locations visited by the corresponding execution, we lift those from being defined for initial states from Σ to also being defined on the set $\Pi \subseteq N^\omega$ of infinite paths in the control flow abstraction (N, E). As we do for initial states, whenever we fix a path $\pi \in \Pi$ (respectively π') we let $(n_i)_{0 \leq i}$ (respectively $(n'_{i'})_{0 \leq i'}$) denote the corresponding sequence of control locations, which we uniformly refer to as the execution in this section, independently of whether it actually stems from an initial state or is a directly fixed path in (N, E). The theorems in this section for paths in the control flow abstraction are directly applicable for initial states when using the original definitions of dependencies and the control slice on initial states.

In addition to our definition of control dependence lifted from initial states, which is transitive as we will prove in Lemma 6.11, we define the intransitive version, the immediate control dependency relation. The immediate control dependency relation consist of only those control dependencies, where the first index is the greatest index upon which the second index is control dependent. This definition also gives directly rise to the corresponding definition for initial states, which we will utilise later.

Definition 6.1 (Immediate control dependency)**.** For any path $\pi \in \Pi$ (or initial state) we define the *immediate control dependency relation* $\xrightarrow{icd}_\pi \subseteq \mathbb{N} \times \mathbb{N}$ by

$$i \xrightarrow{icd}_\pi j \Leftrightarrow i \xrightarrow{cd}_\pi j \wedge \nexists k \in (i, j) \colon k \xrightarrow{cd}_\pi j.$$

As control dependence and thereby the control slice are defined based on the strict post dominance relation ($\xrightarrow{pd}$), we begin with observations about the latter. The following lemmas establish the fact that the terminal control location ($\mathfrak{te}$) strictly post dominates any other control location but is itself not strictly post dominated by any control location.

Lemma 6.1.

$$x \neq \mathfrak{te} \Rightarrow \mathfrak{te} \xrightarrow{pd} x$$

Proof. Any path from x to $\mathfrak{te}$ trivially contains $\mathfrak{te}$, with the antecedent the claim follows. □

Lemma 6.2.

$$\neg\, x \xrightarrow{pd} \mathfrak{te}$$

Proof. In the case where $x = \mathfrak{te}$, we directly obtain that x does not strictly post dominate $\mathfrak{te}$. If $x \neq \mathfrak{te}$ consider the unique path starting at $\mathfrak{te}$ that only contains $\mathfrak{te}$, which exists as $\mathfrak{te}$ is the unique successor of itself in E. Thereby x cannot strictly post dominate $\mathfrak{te}$ as x does not appear on this path. □

Due to our assumption that the terminal control location can be reached from all control locations it follows that the strict post dominance relation is anti-symmetric.

Lemma 6.3.

$$y \xrightarrow{pd} x \Rightarrow \neg\, x \xrightarrow{pd} y$$

Proof. From $y \xrightarrow{pd} x$ we obtain by Lemma 6.2 that $x \neq \mathfrak{te}$. Let π be a path from x to $\mathfrak{te}$, that is $n_0 = x$ and $n_k = \mathfrak{te}$. Such a path exists by Assumption 2 of Definition 4.2. We can assume that x does not occur in π after 0, as it cannot occur after k as $\mathfrak{te}$ only admits self loops and we can shorten π if there is another instance of x. As $y \xrightarrow{pd} x$ and π is a path from x to $\mathfrak{te}$ there exists an index $j \leq k$, such that $n_l = y$. Due to $y \xrightarrow{pd} x$ we have $x \neq y$ and thereby $j > 0$. The subpath from j is then a path from y to $\mathfrak{te}$ that does not contain x and therefore x cannot strictly post dominate y. □

From the definition of the strict post dominance relation and utilising its anti-symmetry, it follows that the strict post dominance relation is transitive.

Lemma 6.4.

$$z \xrightarrow{pd} y \wedge y \xrightarrow{pd} x \Rightarrow z \xrightarrow{pd} x$$

Proof. Assume the antecedent, whereby $x \neq z$ holds by Lemma 6.3. Let π be any path from x to $\mathfrak{te}$, we have to show that π contains z. As $y \xrightarrow{pd} x$ there exists an instance of y on π before $\mathfrak{te}$. The subpath from the instance of y to $\mathfrak{te}$ contains an instance of z before $\mathfrak{te}$ as $z \xrightarrow{pd} y$. Therefore π contains the required instance of z before $\mathfrak{te}$. □

Another fact following from the assumption that all control locations can reach the terminal control location is that any control location can only have finitely many strict post dominators.

Lemma 6.5.

$$|\{y \mid y \xrightarrow{pd} x\}| < \infty$$

Proof. As x reaches $\mathfrak{te}$ there is a path from x to $\mathfrak{te}$, as all strict post dominators of x appear on that path before it reaches $\mathfrak{te}$ after finitely many steps, there can only be finitely many. □

Moving on from the strict post dominance relation itself, we will now investigate basic properties of control dependence and the control slice. A first observation is that while the dependency relations ($\xrightarrow{cd}$, $\xrightarrow{icd}$, $\xrightarrow{dd}$) and the control slice are defined based on whole executions, their definitions actually only depend on the sequence of control locations appearing between the indices they relate, respectively the control locations appearing up to the index for the control slice. We therefore have the following properties which allow us to switch between paths.

Lemma 6.6.

$$\begin{aligned}(\forall l \in [i,j] \colon n_l = n'_{l+k}) \implies \; & (i \xrightarrow{cd}_\pi j \Leftrightarrow i+k \xrightarrow{cd}_{\pi'} j+k) \land \\ & (i \xrightarrow{icd}_\pi j \Leftrightarrow i+k \xrightarrow{icd}_{\pi'} j+k) \land \\ & (i \xrightarrow{dd_v}_\pi j \Leftrightarrow i+k \xrightarrow{dd_v}_{\pi'} j+k)\end{aligned}$$

$$(\forall l \le i \colon n_l = n'_l) \implies \mathrm{cs}_i^\pi = \mathrm{cs}_i^{\pi'}$$

Proof. This follows directly from closely inspecting Definitions 5.2, 5.3, 5.4 and 6.1. □

The following two lemmas characterise the control dependencies and control slice of indices that reached the terminal control location ($\mathfrak{te}$). Firstly, indices that have reached the terminal control location are exactly control dependent upon the smaller indices that also reached $\mathfrak{te}$. Secondly, the control slice of any index reaching $\mathfrak{te}$ thereby only depends upon the number of times the execution has looped there.

Note that this in particular means that any two terminating executions reach matching indices when first reaching the terminal control location. This will be beneficial later, as we will often require indices with matching control slices and we can by assumption at any point modify the sequence of control locations to eventually reach the terminal control location ($\mathfrak{te}$).

Lemma 6.7.

$$n_k = \mathfrak{te} \Rightarrow (i \xrightarrow{cd}_\pi k \Leftrightarrow i < k \land n_i = \mathfrak{te})$$

Proof. If $i \xrightarrow{cd}_\pi k$ we have by Definition 5.2 that $i < k$ and $\neg\, n_k \xrightarrow{pd} n_i$, whereby $n_i = \mathfrak{te}$ by Lemma 6.1.

If $i < k$ and $n_i = \mathfrak{te}$ then it holds that $\forall j \in [i, k] \colon n_j = \mathfrak{te}$ because $\mathfrak{te}$ only reaches itself by Assumption 4 of Definition 4.2. As $\neg\, \mathfrak{te} \xrightarrow{pd} \mathfrak{te}$ we obtain $i \xrightarrow{cd}_\pi k$ as required. □

Lemma 6.8.

$$n_r = \mathfrak{te} \wedge (\forall i < r \colon n_i \neq \mathfrak{te}) \Rightarrow \mathrm{cs}^\pi_{r+k} = (\mathfrak{te})_{i\in[0,k]}$$

Proof. As $n_{r+k} = \mathfrak{te}$ it follows by Lemma 6.7 that $r + k$ is exactly control dependent in π upon the previous instances of $\mathfrak{te}$, which are exactly the ones between r and $r + k$, as r is the first instance of $\mathfrak{te}$ and $\mathfrak{te}$ only reaches itself. □

Besides identifying indices with matching control slices, we also require means to tell indices apart based on their control slice. The following lemma makes a first step in this direction by showing that whenever a control location is reached twice within one execution, there must exist a control dependency for the later visit, which lies at or after the earlier visit. Intuitively this follows from our assumption that according to the control flow abstraction any loop must have the option to terminate. We already used this assumption to prove the transitivity of the post dominance relation, from which the result follows based on our definition of control dependence.

Lemma 6.9.

$$i < j \wedge n_i = n_j \Rightarrow \exists k \in [i, j) \colon k \xrightarrow{cd}_\pi j$$

Proof. Assume the consequent is false. Unfolding Definition 5.2, we thus have

$$\forall k \in [i, j) \colon \exists l \in [k, j] \colon n_l \xrightarrow{pd} n_k. \tag{6.1}$$

We then obtain $\forall k \in [i, j) \colon n_j \xrightarrow{pd} n_k$ by induction over $j - k$ using (6.1) for the base case and (6.1) together with the transitivity of the post dominator relation (Lemma 6.4) in the inductive case. This yields $n_j \xrightarrow{pd} n_i$, which is a contradiction to $n_i = n_j$ and thus the consequent must be true. □

The following two lemmas establish that control dependencies are well-nested, that is if an index is control dependent upon another then so are all intermediate indices and moreover the control dependency relation is transitive.

Lemma 6.10.

$$i \xrightarrow{cd}_\pi k \land i < j \leq k \Rightarrow i \xrightarrow{cd}_\pi j$$

Proof. This follows directly from Definition 5.2 as there cannot exist a strict post dominator of n_i between i and j if there is none between i and k. □

Lemma 6.11.

$$i \xrightarrow{cd}_\pi j \land j \xrightarrow{cd}_\pi k \Rightarrow i \xrightarrow{cd}_\pi k$$

Proof. Assume the consequent is false, we then obtain an index $l \in [i, k]$ such that $n_l \xrightarrow{pd} n_i$. As $i \xrightarrow{cd}_\pi j$ it must be the case that $l \in (j, k]$ and therefore n_l must not post dominate n_j as $j \xrightarrow{cd}_\pi k$. If n_l does not post dominate n_j we can obtain a path from n_j to $\mathfrak{te}$ that does not contain n_l. If we append this path to π at position j and drop the prefix before i we obtain a path from n_i to $\mathfrak{te}$ that does not contain n_l, which is a contradiction as $n_l \xrightarrow{pd} n_i$. □

Transferring these well-nestedness properties to the definition of the control slice yields the following lemma which allows us to decompose the control slice. The lemma states that any non-empty prefix of a control slice is itself the control slice of a control dependency of the index the original control slice corresponded to and these indices build a chain of immediate control dependencies.

Lemma 6.12. *Given any path π and $i \in \mathbb{N}$, let $(i_j)_{j\in[0,k]}$ be the maximal strictly ascending sequence with $i_k = i$ and $\forall j < k\colon i_j \xrightarrow{cd}_\pi i$ as in Definition 5.4, such that $\mathrm{cs}^\pi_i = (n_{i_j})_{j\in[0,k]}$. Then it holds that:*

$$\forall l < k\colon \mathrm{cs}^\pi_{i_l} = (n_{i_j})_{j\in[0,l]} \land i_l \xrightarrow{icd}_\pi i_{l+1}$$

Proof. Let $l < k$ hold. Due to $i_l \xrightarrow{cd}_\pi i$ and Lemma 6.11, any control dependency of i_l is also a control dependency of i, wherefore the maximal strictly increasing sequence of indices that defines the control slice of i_l is a subsequence of $(i_j)_{j\in[0,k]}$. Vice versa, due to Lemma 6.10, any control dependency i_j of i is a control dependency of i_l if and only if $i_j < i_l$, which is the case if and only if $j < l$ due to the sequence $(i_j)_{j\in[0,k]}$ being strictly ascending. It follows that the maximal strictly increasing sequence of indices defining $\mathrm{cs}^\pi_{i_l}$ must be $(i_j)_{j\in[0,l]}$, which yields $\mathrm{cs}^\pi_{i_l} = (n_{i_j})_{j\in[0,l]}$ as required. Moreover, as $l < k$, we have in the same manner that $(i_j)_{j\in[0,l]}$ is the strictly increasing sequence of control dependencies of i_{l+1}. With i_l being the greatest control dependency of i_{l+1}, it follows that there can be no further control dependencies of i_{l+1} after i_l, such that $i_l \xrightarrow{icd}_\pi i_{l+1}$ holds as required. □

As the immediate control dependency of an index must be unique by definition, we can specialise the above lemma for a common use case were we deconstruct the control slice based on an immediate control dependency.

Lemma 6.13.

$$i \xrightarrow{icd}_\pi j \Rightarrow \mathrm{cs}^\pi_j = \mathrm{cs}^\pi_i \cdot n_j$$

Proof. From the antecedent it follows that the control slice of j has length greater one, such that by applying Lemma 6.12 we obtain ι with $\iota \xrightarrow{icd}_\pi j$ and $\mathrm{cs}^\pi_j = \mathrm{cs}^\pi_\iota \cdot n_j$. As there cannot exist two different immediate control dependencies for j, it must be the case that $\iota = i$ and the consequent follows. □

We now work towards Lemma 6.16, one of the central lemmas in this section, which will establish the fact that the control slice determines the order in which control locations are visited by executions, that is the order between indices with matching control slices is fixed. To this end the following lemma first considers the case, where two control locations both appear in two executions, such that the index of the one appearing later in the first execution does not have any control dependencies and neither has the index of the appearance of the other control location in the other execution.

Lemma 6.14.

$$n_i = n'_{i'} \wedge n_j = n'_{j'} \wedge (\nexists k \colon k \xrightarrow{cd}_\pi j) \wedge (\nexists k' \colon k' \xrightarrow{cd}_{\pi'} i') \wedge i < j \Longrightarrow i' < j'$$

Proof. Assume the consequent is false and thus $j' \leq i'$ holds. Consider the path obtained by stitching π' from j' onto π at j. This yields a path $\hat{\pi}$ with a loop on n_i between i and $i' + j - j'$. By Lemma 6.9 we obtain a control dependency k of $i' + j - j'$ in $\hat{\pi}$. If $j < k$ then this control dependency lies completely in the section of $\hat{\pi}$ from π', thus by Lemma 6.6 $k + j' - j$ is a control dependency of i' in π', which is a contradiction. Otherwise, if $k \leq j$ we obtain with Lemma 6.10 that k is a control dependency of j in $\hat{\pi}$, which completely lies in the part of $\hat{\pi}$ from π and thus, again by Lemma 6.6, also arrive at a contradiction. □

Together with the previous lemmas we can inductively prove that for any execution the mapping from indices to their control slice is injective.

Lemma 6.15.

$$i =^{\text{cs}}_{\pi,\pi} j \Longrightarrow i = j$$

Proof. We prove the claim by induction over the length of cs^π_i. As the control slice is never empty we start the induction with the case where $\text{cs}^\pi_i = n_i$. We therefore have $\nexists k \colon k \xrightarrow{cd}_\pi i$ and the same for j. The claim then follows from Lemma 6.14 with $\pi' = \pi$, $i' = j$ and $j' = i$. In the other case obtain with Lemma 6.12 two indices k and l such that $k \xrightarrow{icd}_\pi i$, $l \xrightarrow{icd}_\pi j$ and $\text{cs}^\pi_k \cdot n_i = \text{cs}^\pi_i = \text{cs}^\pi_j = \text{cs}^\pi_l \cdot n_j$. We therefore have $k =^{\text{cs}}_{\pi,\pi} l$ with a shorter control slice and by induction hypothesis obtain $k = l$. If we drop the first k control locations from π we obtain the path $\pi \ll (k+1)$ where, by Lemma 6.6, both $i - k - 1$ and $j - k - 1$ have no control dependencies. As in the base case we obtain $i - k - 1 = j - k - 1$ using Lemma 6.14 and therefore $i = j$ as required. □

We can now prove the central lemma about the order of indices with matching control slices in different executions. Together with the later lemmas about unmatched control slices this lemma is the main motivator for the use of the control slice for the inductive definition of critical executions in Definition 5.5.

Lemma 6.16.

$$i =^{\text{cs}}_{\pi,\pi'} i' \wedge j =^{\text{cs}}_{\pi,\pi'} j' \Longrightarrow (i < j \Leftrightarrow i' < j')$$

Proof. Without loss of generality assume $i < j$. By Lemma 6.15 we have $\text{cs}^\pi_i \neq \text{cs}^\pi_j$. We now consider the following cases for this inequality:

1. cs^π_i is a proper prefix of cs^π_j
2. cs^π_j is a proper prefix of cs^π_i
3. cs^π_i and cs^π_j diverge in the first position
4. cs^π_i and cs^π_j share a common non-empty prefix before they diverge

Case 1. As cs^π_i is a proper prefix of cs^π_j, it is also the case that $\text{cs}^{\pi'}_{i'}$ is a prefix of $\text{cs}^{\pi'}_{j'}$. With Lemma 6.12 and Lemma 6.15 it follows that $i' \xrightarrow{cd}_{\pi'} j'$ and therefore $i' < j'$ as required.

Case 2. This case is empty, as like in Case 1 it would follow that $j \xrightarrow{cd}_\pi i$, which contradicts $i < j$.

Case 3. Let k, l, k' and l' be the respective indices of the first control dependence of i, j, i' and j' or the respective index itself if no such control dependence exists. Due to Lemma 6.11, none of these can posses a control dependence itself. We have $k < l$, as it can neither be the case that $l = k$, as that would contradict the control slices of i and j, which begin with n_k and n_l, diverging in the first position, nor be the case that $l < k \leq i < j$, as this would, by Lemma 6.10, imply that $l \xrightarrow{cd}_\pi k$, which again is a contradiction, as k does not have a control dependence.

As l and k' do not have any control dependence, we obtain from $k < l$ with Lemma 6.14 that $k' < l'$, which leaves us with $i' \geq k' < l' \leq j'$. Again with Lemma 6.10, we see that it cannot be the case that $i' \geq l'$, as this would yield $k' \xrightarrow{cd}_{\pi'} l'$ but l' has no control dependence. Therefore it must be the case that $i' < l' \leq j'$ as required.

Case 4. We consider the common non-empty prefix after which the control slices diverge. By Lemmas 6.12 and 6.15 the prefix is the control slice of an index k in π and of an index k' in π', such that k is the last common control dependency of i and j in π, respectively of i' and j' in π'. We then consider the paths $\hat{\pi} = \pi \ll (k+1)$ and $\hat{\pi}' = \pi' \ll (k'+1)$, where we dropped the first k control locations in π and the first k' control locations in π'. With Lemma 6.6 and Lemma 6.11 we see that we are in the same situation with $\hat{\pi}$, $\hat{\pi}'$, $i-k$, $j-k$, $i'-k'$, $j'-k'$ as with π, π', i, j, i', j' in Case 3. Applying the same argument leaves us with $i' - k' < j' - k'$ and we obtain $i' < j'$ as required.

□

Before we continue with the central lemmas on unmatched control slices, we will first develop two further lemmas about the equivalence of control slices in shifted executions. The following lemma is an inductive generalisation of Lemma 6.13, allowing us to split the control slice of an index at an arbitrary control dependency.

Lemma 6.17.

$$i \xrightarrow{cd}_\pi k \Longrightarrow \mathrm{cs}_k^\pi = \mathrm{cs}_i^\pi \cdot \mathrm{cs}_{k-i-1}^{\pi \ll i+1}$$

Proof. We prove the claim through induction over k. May the claim hold for all smaller k. If $i \xrightarrow{icd}_\pi k$ we obtain by Lemma 6.13 that $\mathrm{cs}_k^\pi = \mathrm{cs}_i^\pi \cdot n_k$. As there exists no control dependency between i and k in π, there is none of $k-i-1$ in $\pi \ll i+1$ and by definition we therefore have $\mathrm{cs}_{k-i-1}^{\pi \ll i+1} = n_{i+1+k-i-1} = n_k$ as required. If there exists another control dependency between i and k, let j be the greatest such, which gives us with Lemma 6.10 that $i \xrightarrow{cd}_\pi j \xrightarrow{icd}_\pi k$. Using Lemma 6.13 for $j \xrightarrow{icd}_\pi k$ and $j-i-1 \xrightarrow{icd}_{\pi \ll i+1} k-i-1$ we obtain with the induction hypothesis that $\mathrm{cs}^\pi_k \overset{6.13}{=} \mathrm{cs}^\pi_j \cdot n_k \overset{\mathrm{IH}}{=} \mathrm{cs}^\pi_i \cdot \mathrm{cs}^{\pi \ll i+1}_{j-i-1} \cdot n_k \overset{6.13}{=} \mathrm{cs}^\pi_i \cdot \mathrm{cs}^{\pi \ll i+1}_{k-i-1}$ as required. □

While the above lemma allows us to split the control slice of an instruction itself at an arbitrary control dependency, we are often mostly interested in whether two control slices of certain indices match or not. To this end the following lemma allows us to reason about whether control slices match when splitting the executions at an arbitrary pair of matching indices, which are not necessarily control dependencies. It states that two indices that are reached after a pair of matching indices match if and only if the shifted indices match when dropping the prefixes that reached the matching indices. This will allow us to drop matched prefixes between executions when looking for matched control slices as we do in our inductive definition of critical matching indices.

Lemma 6.18.

$$k =^{\mathrm{cs}}_{\pi,\pi'} k' \implies (\mathrm{cs}^{\pi \ll k}_n = \mathrm{cs}^{\pi' \ll k'}_{n'} \Leftrightarrow \mathrm{cs}^\pi_{k+n} = \mathrm{cs}^{\pi'}_{k'+n'})$$

Proof. For the implication from right to left we assume $\mathrm{cs}^\pi_{k+n} = \mathrm{cs}^{\pi'}_{k'+n'}$ and let $(i_j)_{j \le \iota}$ and $(i'_j)_{j \le \iota}$ be the strictly ascending families of indices from Definition 5.4 for cs^π_{n+k} and $\mathrm{cs}^{\pi'}_{k'+n'}$. Let η and η' be minimal such that $i_\eta \ge k$ and $i'_{\eta'} \ge k'$. Utilising the definition of the control slice and Lemma 6.6 one recognises that $\mathrm{cs}^{\pi \ll k}_n = (n_{i_{j+\eta}})_{j \le \iota - \eta}$ and $\mathrm{cs}^{\pi' \ll k'}_{n'} = (n'_{i'_{j+\eta'}})_{j \le \iota - \eta'}$. As we have $\forall j \le \iota \colon i_j =^{\mathrm{cs}}_{\pi,\pi'} i'_j$, we obtain by Lemma 6.16 that $\forall j \le \iota \colon i_j \ge k \Leftrightarrow i'_j \ge k'$ and thereby $\eta = \eta'$ and $\mathrm{cs}^{\pi \ll k}_n = \mathrm{cs}^{\pi' \ll k'}_{n'}$ as required.

For the implication from left to right assume $\mathrm{cs}^{\pi \ll k}_n = \mathrm{cs}^{\pi' \ll k'}_{n'}$. We first prove the following property:

$$\forall j < k \colon j \xrightarrow{cd}_\pi k+n \Rightarrow \exists j' < k' \colon j =^{\mathrm{cs}}_{\pi,\pi'} j' \xrightarrow{cd}_{\pi'} k'+n' \tag{6.2}$$

Assume the statement does not hold and fix $j < k$ such that $j \xrightarrow{cd}_\pi k+n$ but $\nexists j' < k' \colon j =^{\mathrm{cs}}_{\pi,\pi'} j' \xrightarrow{cd}_{\pi'} k'+n'$. As $j \xrightarrow{cd}_\pi k+n$ by Lemma 6.10 it holds that $j \xrightarrow{cd}_\pi k$ and as $k =^{\mathrm{cs}}_{\pi,\pi'} k'$ we obtain with Lemma 6.12 an index j' such that $j =^{\mathrm{cs}}_{\pi,\pi'} j' \xrightarrow{cd}_{\pi'} k'$. Let l' be the first control dependence of $k'+n'$ in π' after or at k' or $k'+n'$ itself if none exists. We have that l' cannot be control dependent upon j' and therefore obtain the existence of $i' \in (j', l')$ such that $n'_{i'} \xrightarrow{pd} n'_{j'}$. As $j' \xrightarrow{cd}_{\pi'} k'$ we also obtain $k' < i'$. Therefore l' has no control dependence greater or equal to i' and we obtain by Lemma 6.19 that $n'_{l'} \xrightarrow{pd} n'_{i'}$ and therefore by transitivity that $n'_{l'} \xrightarrow{pd} n'_{j'}$. As $\text{cs}_n^{\pi \ll k} = \text{cs}_{n'}^{\pi' \ll k'}$ and $n'_{l'}$ is part of the latter we obtain the existence of $l \in [k, k+n]$ such that $n_l = n'_{l'}$ but then $n_l \xrightarrow{pd} n_j$, which is a contradiction to $j \xrightarrow{cd}_\pi k+n$, whereby (6.2) must hold.

Due to all assumptions being symmetric in π and π' we obtain by the same argument:

$$\forall j' < k' \colon j' \xrightarrow{cd}_{\pi'} k' + n' \Rightarrow \exists j < k \colon j' =^{\text{cs}}_{\pi',\pi} j \xrightarrow{cd}_\pi k + n \tag{6.3}$$

We now consider two cases. Firstly, if there does not exist a control dependence of $k+n$ in π before k, then by (6.3) neither does one of $k'+n'$ in π' before k' and we obtain $\text{cs}^\pi_{k+n} = \text{cs}_n^{\pi \ll k} = \text{cs}^{\pi' \ll k'} = \text{cs}^{\pi'}_{k'+n'}$ as required.

Secondly, if there exists a control dependence of $k+n$ in π before k, then let j be the greatest such. By (6.2) we obtain the existence of an index j' that is a control dependence of $k'+n'$ in π' before k' with $j =^{\text{cs}}_{\pi,\pi'} j'$. We have that j' is also the greatest control dependence of $k'+n'$ in π' before k' due to (6.2) and Lemma 6.16. We therefore have $\text{cs}^{\pi \ll j+1}_{k+n-j-1} = \text{cs}_n^{\pi \ll k} = \text{cs}_{n'}^{\pi' \ll k'} = \text{cs}^{\pi' \ll j'+1}_{k'+n'-j'-1}$, which yields together with Lemma 6.17 that $\text{cs}^\pi_{k+n} = \text{cs}^\pi_j \cdot cs^{\pi \ll j+1}_{k+n-j-1} = \text{cs}^{\pi'}_{j'} \cdot cs^{\pi' \ll j'+1}_{k'+n'-j'-1} = \text{cs}^{\pi'}_{k'+n'}$ as required. $\square$

We now prepare for the proof of Lemma 6.24, our main lemma about unmatched control slices, which considers the case where the control slice of some index is not matched by another execution but some later index is. In this case the lemma allows us to obtain a matched control dependency of the unmatched index, at which the executions take different branches. The proof will be done by an induction over the control slice of the unmatched index, for which Lemma 6.21 will provide the base case and Lemma 6.23 the inductive case.

The following two lemmas first consider relevant properties of control dependence and post dominance. The first lemma considers the case where the last index in a range of indices has no control dependency within that range and shows that the control location reached at the end post dominates the control locations reached at the beginning. The second lemma shows vice versa that when in a range of indices the last index reaches a post dominator of the control location reached by the first index and that last control location is otherwise not reached within that range, then the last index does not possess any control dependencies within the range.

Lemma 6.19.

$$i < k \land (\nexists j \geq i \colon j \xrightarrow{cd}_\pi k) \implies n_k \xrightarrow{pd} n_i$$

Proof. We perform an induction over $k - i$. As $i < k$ and $\neg\, i \xrightarrow{cd}_\pi k$ we obtain the existence of $j \in (i, k]$ with $n_j \xrightarrow{pd} n_i$. If $j = k$ we are done, otherwise applying the induction hypothesis to $i < j < k$ we obtain $n_k \xrightarrow{pd} n_j$ and therefore $n_k \xrightarrow{pd} n_i$ using the transitivity of the post dominator relation (Lemma 6.4). □

Lemma 6.20.

$$i < k \land n_k \xrightarrow{pd} n_i \land (\nexists j \in (i, k) \colon n_j = n_k) \implies \nexists j \geq i \colon j \xrightarrow{cd}_\pi k$$

Proof. Assume the consequent is false. We thus obtain $j \geq i$ such that $j \xrightarrow{cd}_\pi k$. It is therefore the case that $j \in (i, k)$ and $n_k \neq n_j$ and also that n_k does not post dominate n_j. We can then change π at j to reach $\mathfrak{te}$ without visiting n_k. This is a contradiction to $n_k \xrightarrow{pd} n_i$, as there also is no instance of n_k between i and j. □

The following lemma considers the base case of Lemma 6.24 where an index possesses no control dependencies and a later index is matched by another execution and shows that it cannot be the case that the earlier index is unmatched.

Lemma 6.21.

$$n_0 = n'_0 \land l < m \land m =^{\mathrm{cs}}_{\pi,\pi'} m' \land \mathrm{cs}^\pi_l = n_l \implies \exists l' \colon l =^{\mathrm{cs}}_{\pi,\pi'} l'$$

Proof. If $l = 0$ there is nothing to do as $\mathrm{cs}^\pi_0 = n_0 = n'_0 = \mathrm{cs}^{\pi'}_0$. If $l > 0$ and as l has no control dependencies, we can apply Lemma 6.19 in order to obtain that $n_l \xrightarrow{pd} n_0$. Without loss of generality we may then assume that π and π' both reach $\mathfrak{te}$ after m, respectively after m', as otherwise we could append a path to $\mathfrak{te}$ to both at m and m' and would find that any l' that is control slice equivalent to l in the modified path would also be part of the original path due to Lemma 6.16 as $l < m$ and $m =^{\mathrm{cs}}_{\pi,\pi'} m'$. We therefore obtain that n_l must appear in π'. Let l' be the first instance of n_l in π'. As $n'_{l'} \xrightarrow{pd} n'_0$ we obtain by Lemma 6.20 that $\mathrm{cs}^{\pi'}_{l'} = n'_{l'} = n_l = \mathrm{cs}^\pi_l$ as required. □

Moving towards the inductive case, the following lemma proves that an index that is immediately control dependent upon another index post dominates all intermediate indices.

Lemma 6.22.

$$i \xrightarrow{icd}_{\pi} k \wedge i < j < k \Longrightarrow n_k \xrightarrow{pd} n_j$$

Proof. We perform an induction over $k - j$ for arbitrary j. In the base case if $k = j + 1$ we have that n_k must post dominate n_j because otherwise it would be the case that $j \xrightarrow{cd}_{\pi} k$, which is a contradiction to $i \xrightarrow{icd}_{\pi} k$ and $i < j$. In the inductive case we also utilise that j cannot be a control dependency of k and therefore obtain an index $l \in (j, k]$ such that $n_l \xrightarrow{pd} n_j$. If $l = k$ we are done. Otherwise, we apply the induction hypothesis for l, which yields $n_k \xrightarrow{pd} n_l$, such that with the transitivity of the post dominator relation (Lemma 6.4) we obtain $n_k \xrightarrow{pd} n_j$ as required. □

The following lemma describes the inductive case of Lemma 6.24 and shows that if an index that lies before a matched index is immediately control dependent upon a matched index at which both executions take the same branch, then the original index must also be matched by the second execution.

Lemma 6.23.

$$k \xrightarrow{icd}_{\pi} l \wedge k =^{\mathrm{cs}}_{\pi,\pi'} k' \wedge n_{k+1} = n'_{k'+1} \wedge l < m \wedge m =^{\mathrm{cs}}_{\pi,\pi'} m' \Longrightarrow \exists l' \colon l =^{\mathrm{cs}}_{\pi,\pi'} l'$$

Proof. From $k \xrightarrow{icd}_{\pi} l$ we obtain $\mathrm{cs}^{\pi}_{l} = \mathrm{cs}^{\pi}_{k} \cdot n_l$ by Lemma 6.13.

We first consider the case that $l = k + 1$. In this case we obtain $k' \xrightarrow{icd}_{\pi'} k' + 1$ and thereby also with Lemma 6.13 that $\mathrm{cs}^{\pi'}_{k'+1} = \mathrm{cs}^{\pi'}_{k'} \cdot n'_{k+1} = \mathrm{cs}^{\pi}_{k} \cdot n_{k+1} = \mathrm{cs}^{\pi}_{l}$ as required.

Now consider the case that $l > k+1$. Using Lemma 6.22 we obtain that $n_l \xrightarrow{pd} n_{k+1} = n'_{k'+1}$. We may assume without loss of generality that π and π' both reach $\mathfrak{te}$ after m, respectively after m', as otherwise we could append a path to $\mathfrak{te}$ to both at m and m' and would find that any l' that is control slice equivalent to l in the modified paths would also be part of the original paths due to Lemma 6.16, as $l < m$ and $m =^{\mathrm{cs}}_{\pi,\pi'} m'$.

As $n_l \xrightarrow{pd} n'_{k'+1}$ and π' reaches $\mathfrak{te}$, n_l must appear in π' after $k' + 1$. Let l' be the smallest index greater than $k' + 1$ such that $n'_{l'} = n_l$. With Lemma 6.13 it suffices show that $k' \xrightarrow{icd}_{\pi'} l'$.

We have that $n'_{k'} = n_k \neq \mathfrak{te}$ because $n_l = n_{k+1} = \mathfrak{te}$ would be a contradiction to $n_l \xrightarrow{pd} n_{k+1}$ as $\mathfrak{te}$ has no post dominators according to Lemma 6.2. Therefore, with π' reaching $\mathfrak{te}$ and $\mathfrak{te} \xrightarrow{pd} n'_{k'}$, there exists a smallest index $i' > k'$ such that $n'_{i'} \xrightarrow{pd} n'_{k'}$.

As n_l does not post dominate $n'_{k'} = n_k$ due to $k \xrightarrow{icd}_{\pi} l$, we also have that n_l does not post dominate $n'_{i'}$ due to the transitivity of the post dominator relation and $n'_{i'} \xrightarrow{pd} n'_{k'}$.

This yields that $l' < i'$, as otherwise we could construct a path from $n'_{k'+1}$ to $\mathfrak{te}$ without visiting n_l by changing π' at i' because l' was the first index visiting n_l in π' after $k'+1$.

As i' was the first index visiting a post dominator of $n'_{k'}$ after k', we obtain from $l' < i'$ that $k' \xrightarrow{cd}_{\pi'} l'$.

We also see by using Lemma 6.20 that there is no other control dependence of l' after k', as $k'+1 < l$, $n'_{l'} \xrightarrow{pd} n'_{k'+1}$ and l' was the first instance of n_l after k'. We therefore have $k' \xrightarrow{icd}_{\pi'} l'$ as required. □

The property shown in the following lemma is, together with Lemma 6.16, the major motivation for our utilisation of the control slice in the inductive definition of critical executions. It states that if an index is unmatched by another execution that matches a later index, there must be a matched control dependency of the first index that takes different branches in the two executions.

Lemma 6.24.

$$n_0 = n'_0 \wedge l < m \wedge m =^{\text{cs}}_{\pi,\pi'} m' \wedge (\nexists l' \colon l =^{\text{cs}}_{\pi,\pi'} l')$$
$$\implies \exists k, k' \colon k =^{\text{cs}}_{\pi,\pi'} k' \wedge k \xrightarrow{cd}_{\pi} l \wedge n_{k+1} \neq n'_{k'+1}$$

Proof. Assume that $n_0 = n'_0$, $l < m$, $m =^{\text{cs}}_{\pi,\pi'} m'$ and that

$$\forall k, k' \colon k =^{\text{cs}}_{\pi,\pi'} k' \wedge k \xrightarrow{cd}_{\pi} l \Rightarrow n_{k+1} = n'_{k'+1}. \tag{6.4}$$

We show that there exists an l' such that $l =^{\text{cs}}_{\pi,\pi'} l'$ by induction over the length of cs^{π}_{l} for arbitrary l. In the base case, where $\text{cs}^{\pi}_{l} = n_l$, the existence of the required index l' follows directly from Lemma 6.21.

If $\text{cs}^{\pi}_{l} \neq n_l$ we obtain by Lemma 6.12 an index i with $i \xrightarrow{icd}_{\pi} l$ and $\text{cs}^{\pi}_{l} = \text{cs}^{\pi}_{i} \cdot n_l$. As $i < l < m$ and because any control dependence of i is also a control dependence of l, we have that (6.4) also holds for i instead of l and can apply the induction hypothesis for i to obtain i' such that $i =^{\text{cs}}_{\pi,\pi'} i'$.

From (6.4) we also obtain that $n_{i+1} = n'_{i'+1}$ such that by Lemma 6.23 we obtain the required l' with $l =^{\text{cs}}_{\pi,\pi'} l'$. □

The following lemma generalises the above to indices between matched indices in arbitrary executions by using Lemma 6.18.

Lemma 6.25.

$$i =^{\mathrm{cs}}_{\pi,\pi'} i' \wedge i < l < m \wedge m =^{\mathrm{cs}}_{\pi,\pi'} m' \wedge (\nexists l' \colon l =^{\mathrm{cs}}_{\pi,\pi'} l')$$
$$\implies \exists k, k' \colon i \leq k \wedge k =^{\mathrm{cs}}_{\pi,\pi'} k' \wedge k \xrightarrow{cd}_{\pi} l \wedge n_{k+1} \neq n'_{k'+1}$$

Proof. Applying Lemma 6.18 we obtain that $m - i =^{\mathrm{cs}}_{\pi \ll i, \pi' \ll i'} m - i'$ as well as $\nexists l' \colon l - i =^{\mathrm{cs}}_{\pi \ll i, \pi' \ll i'} l'$. With Lemma 6.24 we obtain the existence of k and k' such that $k =^{\mathrm{cs}}_{\pi \ll i, \pi' \ll i'} k'$, $k \xrightarrow{cd}_{\pi \ll i} l - i$ and $n_{i+k+1} \neq n'_{i'+k'+1}$. Again with Lemma 6.18 we also obtain $k+i =^{\mathrm{cs}}_{\pi,\pi'} k'+i'$ and by Lemma 6.6 also have $k+i \xrightarrow{cd}_{\pi} l$ as required. □

While the previous lemmas considered unmatched indices in executions that later reach matching indices — which, via Lemma 6.8, is always the case for terminating executions — we now show in Lemma 6.28 that in the general case that if an index is unmatched then it is either control dependent upon a matched index that takes a different branch or there exists an earlier matched index such that in the other execution all later indices are control dependent upon the matched index, which corresponds to the case where an execution does not match an index because it entered an infinite loop.

The following lemma makes the observation that there are always infinitely many indices upon which all later indices are control dependent. In the case of non-terminating executions, these would be the loop heads, whose immediate post dominator is never reached by the execution and for terminating executions this follows directly, as they loop at the terminal control location. We already did the main work for this proof and can directly obtain the result from the transitivity of the post dominance relation.

Lemma 6.26.

$$|\{i \mid \forall j > i \colon i \xrightarrow{cd}_{\pi} j\}| = \infty$$

Proof. Assume the claim does not hold and fix an upper bound m. We then have $\forall i > m \colon \exists j > i \colon \neg\, i \xrightarrow{cd}_{\pi} j$, which yields $\forall i > m \colon \exists j > i \colon n_j \xrightarrow{pd} n_i$. We thereby obtain an infinite sequence $(i_j)_{j \geq 0}$ such that $\forall j \colon n_{i_{j+1}} \xrightarrow{pd} n_{i_j}$. This cannot be the case as by Lemma 6.4 we would have $\forall k > j \colon n_{i_k} \xrightarrow{pd} n_{i_j}$, which with Lemma 6.5 would yield the existence of $j < k$ such that $n_{i_j} = n_{i_k}$, which contradicts $n_{i_k} \xrightarrow{pd} n_{i_j}$. □

As mentioned above, terminating runs always reach matching indices. The following lemma utilises this to specialises Lemma 6.24 by assuming that the execution that fails to match an index terminates but does not restrict the other execution. As the other execution can be modified to terminate after reaching the unmatched index, this can be reduced to the previous result.

Lemma 6.27.

$$n_0 = n'_0 \wedge n'_{r'} = \mathfrak{te} \wedge (\nexists l' \colon l =^{\text{cs}}_{\pi,\pi'} l') \implies \exists k, k' \colon k =^{\text{cs}}_{\pi,\pi'} k \wedge k \xrightarrow{cd}_{\sigma} l \wedge n_{k+1} \neq n'_{k'+1}$$

Proof. Without loss of generality let r' be minimal such that $n'_{r'} = \mathfrak{te}$ and let π reach $\mathfrak{te}$ at or after l with r be the corresponding minimal index such that $n_r = \mathfrak{te}$. As r and r' are the first instances of $\mathfrak{te}$, we have $r =^{\text{cs}}_{\pi,\pi'} r'$ by Lemma 6.8. As $\nexists l' \colon l =^{\text{cs}}_{\pi,\pi'} l'$ it must be the case that $n_l \neq \mathfrak{te}$ and therefore $l < r$. With this we obtain the existence of k and k' as required from Lemma 6.24. □

Using this we can prove the final lemma about unmatched indices in this section, which considers general executions and shows that whenever an index in one execution is unmatched by another execution there exists a matched index at which the executions take different branches such that either the unmatched index is control dependent upon this index or all subsequent indices in the other execution are.

Lemma 6.28.

$$\begin{gathered} n_0 = n'_0 \wedge (\nexists l' \colon l =^{\text{cs}}_{\pi,\pi'} l') \implies \\ \exists k, k' \colon k < l \wedge k =^{\text{cs}}_{\pi,\pi'} k' \wedge n_{k+1} \neq n'_{k'+1} \wedge (k \xrightarrow{cd}_{\pi} l \vee \forall j' > k' \colon k' \xrightarrow{cd}_{\pi'} j') \end{gathered}$$

Proof. If there exists r' such that $n'_{r'} = \mathfrak{te}$ the claim follows from Lemma 6.27. Therefore let there be no r' such that $n'_{r'} = \mathfrak{te}$ and without loss of generality assume that π reaches $\mathfrak{te}$ at or after l. Let $M' = \{i' \mid \forall j' > i' \colon i' \xrightarrow{cd}_{\pi'} j'\}$. We now make a case distinction based on whether there exists an index in M' that is unmatched by π.

Assume there exists an index $i' \in M'$ such that $\nexists i \colon i =^{\text{cs}}_{\pi,\pi'} i'$. As π reaches $\mathfrak{te}$, using Lemma 6.27 we obtain k and k' such that $k =^{\text{cs}}_{\pi,\pi'} k'$, $k' \xrightarrow{cd}_{\pi'} i'$ and $n_{k+1} \neq n_{k'+1}$. From $k' \xrightarrow{cd}_{\pi'} i'$ with $i' \in M'$ it follows by Lemmas 6.10 and 6.11 that $\forall j' > k' \colon k' \xrightarrow{cd}_{\pi'} j'$. If it is the case that $k < l$ we are finished as k and k' are as required by the consequent. If on the other hand it holds that $l \leq k$ we can close with Lemma 6.24 as $l < k =^{\text{cs}}_{\pi,\pi'} k'$.

Assume that there exists no index in M' that is unmatched by π. Then we have $\forall i' \in M' \colon \exists i \colon i =^{\text{cs}}_{\pi,\pi'} i'$. As M' is unbounded by Lemma 6.26, we obtain the existence of $i > l$ and i' such that $i =^{\text{cs}}_{\pi,\pi'} i'$ and can again close with Lemma 6.24. □

6.2 Insecurity Contradicts

Exploiting the properties about control dependence and matching control slices proven in the previous section, we now move further towards proving our desired property that executions that break our security property are classified as critical observable executions by Definition 5.6. We split off the last step that is done in Definition 5.6 over the inductive definition of critical matching pairs (Definition 5.5) and to this end define the concept of contradicting executions, which do not care about the inductive tracking of data and control dependencies but directly compare executions at arbitrary indices with matching control slices.

We will first prove that executions which break our security property are contradicting executions. This follows mainly from the fact that the control slice determines the order of operations in executions, so that if the observations in two executions differ then there either exists an unmatched index in one execution that provides us with a diverging control dependency through the lemmas in the previous section, or we directly have matching indices that reached an observable control location that reads different data in the two executions. In the next section we will then conclude with the inductive proof that contradicting executions are also classified as critical by our definitions.

Definition 6.2 (Contradicting execution)**.** We say that proper matching of an index i in the execution from σ by the execution from σ' is *contradicted* by an index i', denote by $i\ \mathfrak{c}_{\sigma,\sigma'}\ i'$, as defined through

$$i\ \mathfrak{c}_{\sigma,\sigma'}\ i' \ \Leftrightarrow\ (i =^{\mathrm{cs}}_{\sigma,\sigma'} i' \wedge \sigma_i \neq_{\mathrm{use}(n_i)} \sigma'_{i'}) \ \vee\ (\exists \iota\colon \iota \xrightarrow{cd}_{\sigma} i \wedge \iota =^{\mathrm{cs}}_{\sigma,\sigma'} i' \wedge n_{\iota+1} \neq n'_{i'+1}).$$

The idea behind this definition is that σ' fails to properly match the execution of n_i at i in σ because either there already exists a corresponding execution of n_i at the matching index i' in σ' but the corresponding state $\sigma'_{i'}$ differs from σ_i in a variable read by n_i, or the execution in σ' takes a different branch than the execution in σ at a control dependency of i, which prohibits the proper matching of i by σ'.[2]

[2]In general taking a different branch at a control dependency does not fully rule out that a matching index may still be reached. This is possible if some but not all branches of a branching control location join early and the control location of the to be matched index is common to these branches. If however branches do not join early, which is they only join at the immediate post dominator of the branching control location, for instance because all branches are binary as in our command language, this is not possible.

For convenience we let in the following $(o_i)_{i<\bar{o}}$ (respectively $(o'_i)_{i<\bar{o}'}$) denote the strictly ascending family of observable indices in the execution from σ (respectively σ'), that is used in Definition 4.4 and fulfils $\text{obs}(\sigma) = (\text{obs}(n_{o_i})(\sigma_{o_i}))_{i<\bar{o}}$.

Our definition of security (Definition 4.5) requires that the sequences of observations produced for low equivalent input states must be comparable, which means that there must be no point in the sequences of observations where different values are observed, that is there must be no $k < \min(\bar{o}, \bar{o}')$ such that $\text{obs}(n_{o_k})(\sigma_{o_k}) \neq \text{obs}(n'_{o'_k})(\sigma'_{o'_k})$. We distinguish two ways how a different observed value can be produced at a point in the observation sequence, either at matching indices (that is $o_k =^{\text{cs}}_{\sigma,\sigma'} o'_k$) where the observed value is produced by states that, based on Assumption 6 of Definition 4.2, must differ in the variables read at the reached control location (that is $\sigma_{o_k} \neq_{\text{use}(n_{o_k})} \sigma'_{o'_k}$), or the observable indices corresponding to the point in the observation sequence do not match at all (that is $o_k \neq^{\text{cs}}_{\sigma,\sigma'} o'_k$). The latter case requires, as we will prove, that a matched control dependency of an observable index takes different branches in the two executions, whereby these two cases correspond to the two cases distinguished in our definition of contradicting executions.

We will consider the second case first and to this end the following lemma first makes a note about non-matching observable indices. It states that if at some point in the observation sequence of two executions the observations are produced by non-matching indices then there must be some observable index missing from one of the executions leading to that point, which follows as the control slice determines the order in which the observations can happen.

Lemma 6.29.

$$o_k \neq^{\text{cs}}_{\sigma,\sigma'} o'_k \implies \exists l \leq k \colon (\nexists j' \colon o_l =^{\text{cs}}_{\sigma,\sigma'} j') \vee (\nexists j \colon j =^{\text{cs}}_{\sigma,\sigma'} o'_l)$$

Proof. Without loss of generality let k be minimal such that the antecedent holds and assume that the consequent is false. We therefore obtain for $l = k$ the existence of j and j' such that $o_k =^{\text{cs}}_{\sigma,\sigma'} j'$ and $j =^{\text{cs}}_{\sigma,\sigma'} o'_k$. As o_k and o'_k are observable so are j and j' and we obtain the existence of i and i', such that $o_k =^{\text{cs}}_{\sigma,\sigma'} o'_{i'}$ and $o_i =^{\text{cs}}_{\sigma,\sigma'} o'_k$. Using Lemma 6.16 and the strict monotonicity of $(o_i)_{i<\bar{o}}$ and $(o'_i)_{i<\bar{o}'}$ we obtain $k < i \Leftrightarrow i' < k$. As the indices cannot coincide, either i or i' must be smaller than k. Without loss of generality let $i < k$ hold. As k was minimal we have $o'_i =^{\text{cs}}_{\sigma',\sigma} o_i =^{\text{cs}}_{\sigma,\sigma'} o'_k$, which due to Lemma 6.15 is a contradiction as $o'_i \neq o'_k$. □

Combined with the lemmas from the previous section about unmatched indices, this yields that if at some point in two observation sequences the observations are produced by non-matching indices, then there must exist some observable index in one of the executions that is contradicted by an index in the other execution.

Lemma 6.30.

$$o_k \neq^{\text{cs}}_{\sigma,\sigma'} o'_k \implies \exists l, i \colon o_l \mathfrak{c}_{\sigma,\sigma'} i \vee o'_l \mathfrak{c}_{\sigma',\sigma} i$$

Proof. Without loss of generality we may assume that k is minimal with this property and thereby that $\nexists j' \colon o_k =^{\text{cs}}_{\sigma,\sigma'} j'$, which we obtain from Lemma 6.29 after swapping σ and σ' accordingly.

From Lemma 6.28 we obtain the existence of i and i' such that $i < o_k$, $i =^{\text{cs}}_{\sigma,\sigma'} i'$, $n_{i+1} \neq n'_{i'+1}$ and either $i \xrightarrow{cd}_\sigma o_k$ or $\forall j' > i' \colon i' \xrightarrow{cd}_{\sigma'} j'$. In the former case we directly obtain from Definition 6.2 that $o_k \mathfrak{c}_{\sigma,\sigma'} i'$ as required.

Let it now be the case that $\forall j' > i' \colon i' \xrightarrow{cd}_{\sigma'} j'$. If $o'_k > i'$ we again directly obtain $o'_k \mathfrak{c}_{\sigma',\sigma} i$.

If $o'_k \leq i'$ then it cannot be the case that there exists an index j such that $j =^{\text{cs}}_{\sigma,\sigma'} o'_k$ as this would imply the existence of an index l such that $j = o_l$ and thereby $o_l =^{\text{cs}}_{\sigma,\sigma'} o'_k \leq i' =^{\text{cs}}_{\sigma',\sigma} i < o_k$, which with Lemma 6.16 implies $o_l < o_k$ and thereby $l < k$, which contradicts the minimality of k.

We therefore have that $\nexists j \colon j =^{\text{cs}}_{\sigma,\sigma'} o'_k$ and again apply Lemma 6.28 to obtain the existence of indices ι and ι', such that $\iota' < o'_k$, $\iota =^{\text{cs}}_{\sigma,\sigma'} \iota'$, $n_{\iota+1} \neq n'_{\iota'+1}$ and either $\iota' \xrightarrow{cd}_{\sigma'} o'_k$ or $\forall j > \iota \colon \iota \xrightarrow{cd}_\sigma j$. Again in the former case we obtain $o'_k \mathfrak{c}_{\sigma',\sigma} \iota$ as required. In the latter case we have $\iota =^{\text{cs}}_{\sigma,\sigma'} \iota' < o'_k \leq i' =^{\text{cs}}_{\sigma',\sigma} i < o_k$ and therefore $\iota < o_k$ by Lemma 6.16 and thereby $\iota \xrightarrow{cd}_\sigma o_k$, which yields $o_k \mathfrak{c}_{\sigma,\sigma'} \iota'$ as required. □

With this we can now prove the central theorems linking insecure executions to contradicting executions. The following theorem first considers arbitrary executions where the observation sequences are not comparable and shows that there exists an observable index in one of the executions that is contradicted by an index in the other.

Theorem 6.31.

$$\text{obs}(\sigma) \not\lesssim \text{obs}(\sigma') \not\lesssim \text{obs}(\sigma) \Rightarrow \exists l, i \colon o_l \mathfrak{c}_{\sigma,\sigma'} i \vee o'_l \mathfrak{c}_{\sigma',\sigma} i$$

Proof. From the antecedent we obtain the existence of a position $k < \min(\bar{o}, \bar{o}')$ in the observation sequences such that $\text{obs}(n_{o_k})(\sigma_{o_k}) \neq \text{obs}(n'_{o'_k})(\sigma'_{o'_k})$. We then consider two cases based on whether $o_k =^{\mathrm{cs}}_{\sigma,\sigma'} o'_k$ holds or not. If $o_k \neq^{\mathrm{cs}}_{\sigma,\sigma'} o'_k$ holds then the claim follows directly from Lemma 6.30.

Otherwise, if $o_k =^{\mathrm{cs}}_{\sigma,\sigma'} o'_k$ holds we have $n_{o_k} = n'_{o_k}$. With Assumption 6 of Definition 4.2 and $\mathrm{obs}(n_{o_k})(\sigma_{o_k}) \neq \mathrm{obs}(n'_{o'_k})(\sigma'_{o'_k})$ we obtain $\sigma_{o_k} \neq_{\mathrm{use}(n_{o_k})} \sigma'_{o'_k}$. From this $o_k \,\mathfrak{c}_{\sigma,\sigma'}\, o'_k$ follows by Definition 6.2. □

Finally, together with Lemma 6.27 from the previous section this also enables us to prove the corresponding theorem for terminating executions, for which the following theorem shows that if the observation sequence of an arbitrary execution is not a prefix of the observation sequence of a terminating execution then there exists an observable index in one of the executions that is contradicted by an index in the other.

Theorem 6.32.

$$n_r = \mathfrak{te} \wedge \mathrm{obs}(\sigma') \not\leq \mathrm{obs}(\sigma) \Rightarrow \exists k, i \colon o'_k \,\mathfrak{c}_{\sigma',\sigma}\, i \vee o_k \,\mathfrak{c}_{\sigma,\sigma'}\, i$$

Proof. We first consider the case where some observable index in σ' is not matched by σ that is we assume there exists $l < \bar{o}'$ such that $\nexists j \colon j =^{\mathrm{cs}}_{\sigma,\sigma'} o'_l$. By Lemma 6.27 and $n_r = \mathfrak{te}$ we obtain the existence of indices ι and ι' such that $\iota =^{\mathrm{cs}}_{\sigma,\sigma'} \iota' \xrightarrow{cd}_{\sigma'} o'_l$ and $n_{\iota+1} \neq n'_{\iota'+1}$ whereby we obtain $o'_l \,\mathfrak{c}_{\sigma',\sigma}\, \iota$ by Definition 6.2 as required.

Otherwise, we have $\forall l < \bar{o}' \colon \exists j \colon j =^{\mathrm{cs}}_{\sigma,\sigma'} o'_l$ and together with the fact that the o'_l are distinct and the control slice is injective we obtain that $\bar{o}' \leq \bar{o}$. This yields together with $\mathrm{obs}(\sigma') \not\leq \mathrm{obs}(\sigma)$ the existence of a position $l < \bar{o}'$ such that $\mathrm{obs}(n_{o_l})(\sigma_{o_l}) \neq \mathrm{obs}(n'_{o'_l})(\sigma_{o'_l})$, whereby $\mathrm{obs}(\sigma) \not\leq \mathrm{obs}(\sigma')$ holds and the claim follows by Theorem 6.31.
□

6.3 Contradictions are Critical

While the previous section provided the connection between security and contradicting executions, we now move to proving that contradicting executions are captured by our inductive definition of critical executions. The main preparations for this were already done in Section 6.1 in the form of the ordering lemma for control slices (Lemma 6.16) and the lemmas on unmatched control slices. Most of the remaining work is done by the following theorem through induction and careful case distinction.

Theorem 6.33.

$$\sigma =_L \sigma' \wedge k =^{\text{cs}}_{\sigma,\sigma'} k' \wedge \sigma_k \neq_{\text{use}(n_k)} \sigma'_{k'} \Longrightarrow k \bowtie_{\sigma,\sigma'} k'$$

Proof. We prove the claim through induction over the sum of k and k'. Our induction hypothesis thus is that the claim holds for any $\hat{k}$, $\hat{k}'$ such that $\hat{k} + \hat{k}' < k + k'$.

First consider the case that $k = k' = 0$. By Assumption 7 of Definition 4.2 we have $H = \text{use}(\mathfrak{st}) = \text{use}(n_0)$ whereby it holds that $\sigma = \sigma_0 \neq_H \sigma'_0 = \sigma'$ and therefore with the base case (5.3) of Definition 5.5 we obtain $0 \bowtie_{\sigma,\sigma'} 0$ as required.

Now let k and k' be greater than zero. Note that due to $0 =^{\text{cs}}_{\sigma,\sigma'} 0$ and Lemma 6.15, it cannot be the case that only one of them is greater than zero. Let $v \in \text{use}(n_k)$ be such that $\sigma_k(v) \neq \sigma'_{k'}(v)$. As the values of the low variables L coincide initially and all other variables are written at the initial control location $\mathfrak{st} = n_0 = n'_0$, we obtain that there must exist at least one index smaller than k where v is written in the first execution or an index smaller than k' where v was written in the second execution. Therefore there must exist at least one data dependency of v at k in the first execution or at k' in the second execution. We make a case distinction based on whether v at k and k' is data dependent upon matching indices in the two executions that is we consider the cases:

1. $j =^{\text{cs}}_{\sigma,\sigma'} j' \wedge j \xrightarrow{dd_v}_{\sigma} k \wedge j' \xrightarrow{dd_v}_{\sigma'} k'$,
2. $\nexists j, j' \colon j =^{\text{cs}}_{\sigma,\sigma'} j' \wedge j \xrightarrow{dd_v}_{\sigma} k \wedge j' \xrightarrow{dd_v}_{\sigma'} k'$

Case 1. We have $\sigma_{j+1}(v) = \sigma_k(v) \neq \sigma'_{k'}(v) = \sigma'_{j'+1}(v)$ and $v \in \text{def}(n_j)$ therefore by Assumption 6 of Definition 4.2 we have $\sigma_j \neq_{\text{use}(n_j)} \sigma_{j'}$. With $j + j' < k + k'$ we obtain from the induction hypothesis that $j \bowtie_{\sigma,\sigma'} j'$, which together with $(j, j) \xrightarrow{dd_v}_{\sigma,\sigma'} (k, k')$ and rule (5.4) from the inductive definition of $\bowtie$ yields $k \bowtie_{\sigma,\sigma'} k'$ as required.

Case 2. It cannot be the case that there exist i, i', j and j' such that $i =^{\text{cs}}_{\sigma,\sigma'} i'$, $j =^{\text{cs}}_{\sigma,\sigma'} j'$, $i \xrightarrow{dd_v}_{\sigma} k$ and $j' \xrightarrow{dd_v}_{\sigma'} k'$ but $i \neq j$ due to Lemma 6.16. Therefore there exists at least one unmatched data dependency of v at k or at k' in the corresponding execution that is we have

$$(\exists j \colon j \xrightarrow{dd_v}_{\sigma} k \wedge \nexists j' \colon j =^{\text{cs}}_{\sigma,\sigma'} j') \vee (\exists j' \colon j' \xrightarrow{dd_v}_{\sigma'} k' \wedge \nexists j \colon j =^{\text{cs}}_{\sigma,\sigma'} j').$$

With Lemma 6.24 we obtain from $k =^{\text{cs}}_{\sigma,\sigma'} k'$ that there must exist at least one diverging control dependency of a data dependency of v at k or k' in the corresponding execution that is

$$\begin{gathered}\exists i, i' \colon i =^{\text{cs}}_{\sigma,\sigma'} i' \wedge n_{i+1} \neq n'_{i'+1} \wedge \\ ((\exists j \colon i \xrightarrow{cd}_{\sigma} j \xrightarrow{dd_v}_{\sigma} k \wedge \nexists j' \colon j =^{\text{cs}}_{\sigma,\sigma'} j') \vee \\ (\exists j' \colon i' \xrightarrow{cd}_{\sigma'} j' \xrightarrow{dd_v}_{\sigma'} k' \wedge \nexists j \colon j =^{\text{cs}}_{\sigma,\sigma'} j')).\end{gathered} \tag{6.5}$$

As any such i is bound by k due to Lemma 6.16, we can fix i and i' as above such that i is maximal with this property. It follows that i' is also maximal with this property due to Lemma 6.16.

Assume that it is the case that there exists an index j such that $i \xrightarrow{cd}_{\sigma} j \xrightarrow{dd_v}_{\sigma} k$ and $\nexists j' \colon j =^{\text{cs}}_{\sigma,\sigma'} j'$. The other case works analogously.

As $n_i = n'_{i'}$ but $n_{i+1} \neq n'_{i'+1}$ it must be the case that $\sigma_i \neq_{\text{use}(n_i)} \sigma'_{i'}$ due to Assumption 6 of Definition 4.2. As $i + i' < k + k'$ due to Lemma 6.16, we obtain from the induction hypothesis that $i \bowtie_{\sigma,\sigma'} i'$.

In order to obtain $(i, i') \xrightarrow{dcd_v}_{\sigma,\sigma'} (k, k')$ through (5.2) and thereby $k \bowtie_{\sigma,\sigma'} k'$ through (5.4), what remains to be shown is that

$$\forall l, l', \iota' \colon i < l =^{\text{cs}}_{\sigma,\sigma'} l' \leq \iota' < k' \Rightarrow v \notin \text{def}(n'_{\iota'}). \tag{6.6}$$

Assume that (6.6) does not hold and fix l,l' and ι' accordingly such that $i < l =^{\text{cs}}_{\sigma,\sigma'} l' < k'$ holds and ι' is maximal in $[l', k')$ with $v \in \text{def}(n'_{\iota'})$. As ι' is chosen maximal we have that $\iota' \xrightarrow{dd_v}_{\sigma'} k'$.

It must therefore be the case that there exists ι with $\iota =^{\text{cs}}_{\sigma,\sigma'} \iota'$ because i' is maximal with property (6.5) and otherwise Lemma 6.25 would yield a diverging control dependency of ι' that lies at or after l' and thereby is greater than i' but also has property (6.5), contradicting the maximality of i'.

As $\iota < k$, $v \in \text{def}(n_\iota)$ and $j \xrightarrow{dd_v}_{\sigma} k$, it must be the case that $\iota \leq j$ and as j is unmatched even that $\iota < j$. But then Lemma 6.25 again yields a diverging control dependency of j that is greater or equal to ι and thereby greater than i but also has property (6.5), which is a contradiction to i being maximal with that property. We therefore have that (6.6) must hold as required. □

The above theorem corresponds to one implication of the to be proven Theorem 5.1 from Chapter 5, which follows as the other implication holds by induction.

Corollary 6.34 (Theorem 5.1)**.**

$$\sigma =_L \sigma' \wedge i =^{\mathrm{cs}}_{\sigma,\sigma'} i' \wedge \sigma_i \neq_{\mathrm{use}(n_i)} \sigma'_{i'} \iff i \bowtie_{\sigma,\sigma'} i'$$

Proof. The implication from left to right is given by Theorem 6.33 and the implication from right to left follows by induction from Definition 5.5. It holds for the base case (5.3), due to the fact that $0 \bowtie_{\sigma,\sigma'} 0$ implies $\sigma =_L \sigma'$ and $\sigma \neq_H \sigma'$, which with $\mathrm{use}(\mathfrak{st}) = H$ implies $\sigma_0 \neq_{\mathrm{use}(n_0)} \sigma'_0$ and $0 =^{\mathrm{cs}}_{\sigma,\sigma'} 0$ holds for all initial states. In the inductive case (5.4), where $\sigma =_L \sigma'$ follows from the induction hypothesis, $i =^{\mathrm{cs}}_{\sigma,\sigma'} i'$ holds by assumption and we have a difference in a variable that is read at the reached control location, as at least one index is the target of a data dependency on that variable. □

As our definition of contradicting executions was especially tailored to bridge the gap between our notions of critical executions (Definition 5.5) and critical observable executions (Definition 5.6), we can obtain from the above result about critical executions that contradicting executions that reach observable locations from low equivalent inputs are captured by our notion of critical observable executions.

Theorem 6.35.

$$\sigma =_L \sigma' \wedge k \mathrel{\mathfrak{c}_{\sigma,\sigma'}} k' \wedge n_k \in \mathrm{dom}(\mathrm{obs}) \iff k \ltimes_{\sigma,\sigma'} k'$$

Proof. For the implication from left to right assume that $\sigma =_L \sigma'$, $k \mathrel{\mathfrak{c}_{\sigma,\sigma'}} k'$ and $n_k \in \mathrm{dom}(\mathrm{obs})$ hold. From $k \mathrel{\mathfrak{c}_{\sigma,\sigma'}} k'$ we obtain by Definition 6.2 that either $k =^{\mathrm{cs}}_{\sigma,\sigma'} k'$ and $\sigma_k \neq_{\mathrm{use}(n_k)} \sigma'_{k'}$ or $\exists i\colon i \xrightarrow{cd}_\sigma k \wedge i =^{\mathrm{cs}}_{\sigma,\sigma'} k' \wedge n_{i+1} \neq n'_{k'+1}$ hold.

If it is the case that $k =^{\mathrm{cs}}_{\sigma,\sigma'} k'$ and $\sigma_k \neq_{\mathrm{use}(n_k)} \sigma'_{k'}$ hold, then we obtain by Theorem 6.33 that $k \bowtie_{\sigma,\sigma'} k'$. Together with $n_k \in \mathrm{dom}(\mathrm{obs})$ we have by Definition 5.6 that $k \ltimes_{\sigma,\sigma'} k'$ as required.

Otherwise, if $i \xrightarrow{cd}_\sigma k$, $i =^{\mathrm{cs}}_{\sigma,\sigma'} k'$ and $n_{i+1} \neq n'_{k'+1}$ hold, then by Assumption 6 of Definition 4.2 we have that $\sigma_i \neq_{\mathrm{use}(n_i)} \sigma'_{k'}$ and therefore again by Theorem 6.33 obtain $i \bowtie_{\sigma,\sigma'} k'$ and again with Definition 5.6, as $i \xrightarrow{cd}_\sigma k$ and $n_{i+1} \neq n'_{k'+1}$, find that $k \ltimes_{\sigma,\sigma'} k'$ holds as required.

For the other implication assume that $k \ltimes_{\sigma,\sigma'} k'$ holds. By Definition 5.6 we directly obtain that $n_k \in \mathrm{dom}(\mathrm{obs})$ and that either $k \bowtie_{\sigma,\sigma'} k'$ or there exists ι such that $\iota \xrightarrow{cd}_\sigma k$, $\iota =^{\mathrm{cs}}_{\sigma,\sigma'} k'$ and $n_{\iota+1} \neq n_{k'+1}$. Depending on which case holds let i be either k itself, if $k \bowtie_{\sigma,\sigma'} k'$ holds and otherwise be the presumed ι. We therefore have in either case $i \bowtie_{\sigma,\sigma'} k'$ and obtain from Corollary 6.34 that $\sigma =_L \sigma'$, $i =^{\mathrm{cs}}_{\sigma,\sigma'} k'$ and $\sigma_i \neq_{\mathrm{use}(n_i)} \sigma'_{k'}$. The two cases now directly correspond to the two cases in Definition 6.2 such that we obtain $k\ \mathfrak{c}_{\sigma,\sigma'}\ k'$ as required. □

Combining the above result, which links critical observable executions to contradicting executions, with the results from the previous section that link contradicting executions to executions that break our security property, we finally obtain our desired soundness properties for our notion of critical observable executions.

Firstly, in combination with Theorem 6.31 we obtain Part b) of Theorem 5.2, which states that if the observation sequences for two low equivalent input states are not comparable then there exists a critical observable index in at least one of the executions as witnessed by an index in the other execution.

Corollary 6.36 (Part b of Theorem 5.2)**.**

$$\sigma =_L \sigma' \wedge \mathrm{obs}(\sigma) \not\lesssim \mathrm{obs}(\sigma') \wedge \mathrm{obs}(\sigma') \not\lesssim \mathrm{obs}(\sigma) \Rightarrow \exists i, i' \colon i \ltimes_{\sigma,\sigma'} i' \vee i' \ltimes_{\sigma',\sigma} i$$

Proof. By Theorem 6.31 we obtain k and i such that $o_k\ \mathfrak{c}_{\sigma,\sigma'}\ i$ or $o'_k\ \mathfrak{c}_{\sigma',\sigma}\ i$ and conclude by Theorem 6.35 that $o_k \ltimes_{\sigma,\sigma'} i$ or $o'_k \ltimes_{\sigma',\sigma} i$ holds. □

Secondly, in combination with Theorem 6.32 we obtain Part a) of Theorem 5.2, which considers terminating executions and states that if for an execution there exist another terminating execution from a low equivalent input state such that the observation sequence of the former is not a prefix of the latter then there exists a critical observable index in at least of one of the executions that is witnessed by the other.

Corollary 6.37 (Part a of Theorem 5.2)**.**

$$\sigma =_L \sigma' \wedge \mathrm{obs}(\sigma') \not\lesssim \mathrm{obs}(\sigma) \wedge (\exists r \colon n_r = \mathfrak{te}) \Longrightarrow \exists i, i' \colon i \ltimes_{\sigma,\sigma'} i' \vee i' \ltimes_{\sigma',\sigma} i$$

Proof. From Theorem 6.32 we obtain k and i such that $o_k\ \mathfrak{c}_{\sigma,\sigma'}\ i$ or $o'_k\ \mathfrak{c}_{\sigma',\sigma}\ i$ hold and obtain from Theorem 6.35 that $o_k \ltimes_{\sigma,\sigma'} i$ or $o'_k \ltimes_{\sigma',\sigma} i$ hold as required. □

6.4 Critical Reaching Executions

We conclude this chapter with the proof of Theorem 5.4, the correctness result for our single execution approximation, which we called critical reaching executions. Our notion of (critical) reaching executions from Definition 5.7 mostly corresponds to the definition of critical execution pairs in Definition 5.5, yet the inductive rules in both definitions do not completely match one-to-one. The definition of data control dependencies on single executions in equation (5.5) of Definition 5.7 is somewhat stricter than the corresponding definition of data control dependencies on pairs of executions in equation (5.2) of Definition 5.5, as the former does not allow any writes to the variable in question, while the latter only forbids them after the executions have reached matching indices for the first time. For (critical) reaching executions these dependencies are already captured though the other dependencies. The following lemma establishes that our notion of critical pairs ($\bowtie$) is safely approximated by our notion of reaching executions (R).

Lemma 6.38.

$$k \bowtie_{\sigma,\sigma'} k' \Rightarrow (\sigma, k) \in R \wedge (\sigma', k') \in R$$

Proof. We prove the claim via induction over the inductive definition of $k \bowtie_{\sigma,\sigma'} k'$.

In the base case (5.3) where $0 \bowtie_{\sigma,\sigma'} 0$ holds, we directly have $(\sigma, 0) \in R$ and $(\sigma', 0) \in R$ by the first rule (5.6) of Definition 5.7.

Now let $k \bowtie_{\sigma,\sigma'} k'$ hold via the inductive rule (5.4) of Definition 5.5. We therefore have $k =^{\text{cs}}_{\sigma,\sigma'} k'$, $\sigma_k(v) \neq \sigma'_{k'}(v)$ for some $v \in \text{Var}$ and the existence of i and i' with $i \bowtie_{\sigma,\sigma'} i'$ such that $(i, i') \xrightarrow{dd_v}_{\sigma,\sigma'} (k, k')$ or $(i, i') \xrightarrow{dcd_v}_{\sigma,\sigma'} (k, k')$ or $(i', i) \xrightarrow{dcd_v}_{\sigma',\sigma} (k', k)$. From the induction hypothesis we have $(\sigma, i) \in R$ and $(\sigma', i') \in R$.

Consider the first case $(i, i') \xrightarrow{dd_v}_{\sigma,\sigma'} (k, k')$. As $i \xrightarrow{dd_v}_{\sigma} k$ and $i' \xrightarrow{dd_v}_{\sigma'} k'$ we obtain by the second rule of Definition 5.7 that $(\sigma, k) \in R$ and $(\sigma', k') \in R$ as required.

As the second and third case are symmetrical we only consider the second, the other works analogously. Let it be the case that $(i, i') \xrightarrow{dcd_v}_{\sigma,\sigma'} (k, k')$ holds. From this we obtain with (5.2) that $n_{i+1} \neq n'_{i'+1}$ as well as the existence of an index j such that $i \xrightarrow{cd}_{\sigma} j \xrightarrow{dd_v}_{\sigma} k$ and that $\forall l, l', j' \colon i < l =^{\text{cs}}_{\sigma,\sigma'} l' \leq j' < k' \Rightarrow v \notin \text{def}(n'_{j'})$. As $(\sigma, i) \in R$ and $i \xrightarrow{cd}_{\sigma} j \xrightarrow{dd_v}_{\sigma} k$ we obtain from rules (5.7) and (5.8) of Definition 5.7 that $(\sigma, j) \in R$ and $(\sigma, k) \in R$. What remains to be shown is that $(\sigma', k') \in R$. We make a case distinction based on whether there exists an index $j' \in [i', k')$ such that $v \in \text{def}(n'_{j'})$.

If there is no such j' then, as $i \bowtie_{\sigma,\sigma'} i'$ implies $i =^{\text{cs}}_{\sigma,\sigma'} i' \wedge \sigma =_L \sigma'$ and we already obtained $k =^{\text{cs}}_{\sigma,\sigma'} k'$ and $\sigma_k(v) \neq \sigma'_{k'}(v)$, we have by (5.5) that $i' \xrightarrow{dcd_v^{\sigma,i}}_{\sigma'} k'$. Together with $(\sigma, i) \in R$ and $(\sigma', i') \in R$ we obtain the required $(\sigma', k') \in R$ using rule (5.9) of Definition 5.7.

Now consider the case where we have an index $j' \in [i', k')$ such that $v \in \text{def}(n'_{j'})$ and let j' be the greatest such. We therefore have that $j' \xrightarrow{dd_v}_{\sigma'} k'$ and can close the proof by showing that $(\sigma', j') \in R$ holds. If $j' = i'$ holds we are done. Therefore assume that $i' < j'$. From $\forall l, l', j' \colon i < l =^{\text{cs}}_{\sigma,\sigma'} l' \leq j' < k' \Rightarrow v \notin \text{def}(n'_{j'})$, utilising $i' \leq j' < k'$, $v \in \text{def}(n'_{j'})$ and Lemma 6.16, we obtain $(*)$: $\forall l' \in (i', j'] \colon \nexists l \colon l =^{\text{cs}}_{\sigma,\sigma'} l'$. With $i' < j'$ it follows that j' must be unmatched by σ. Using Lemma 6.25 we then obtain the existence of indices ι and ι' such that $i' \leq \iota'$, $\iota' \xrightarrow{cd}_{\sigma'} j'$ and $\iota =^{\text{cs}}_{\sigma,\sigma'} \iota'$. Due to $(*)$ it must be the case that $\iota' = i'$ and thereby we obtain the required $(\sigma', j') \in R$ using rule (5.8) of Definition 5.7. ☐

The final step that links critical reaching executions to critical observable pairs is straightforward by unfolding the definitions and applying the above result.

Corollary 6.39 (Theorem 5.4)**.**

$$i \ltimes_{\sigma,\sigma'} i' \quad \Longrightarrow \quad (\sigma, i) \in C \wedge (\sigma', i') \in R$$

Proof. From Definition 5.6 we have that $n_i \in dom(\text{obs})$ and either $i \bowtie_{\sigma,\sigma'} i'$ or obtain the existence of an index ι such that $\iota \xrightarrow{cd}_{\sigma} i$ and $\iota \bowtie_{\sigma,\sigma'} i'$.

In the former case we obtain $(\sigma, i) \in R$ and $(\sigma', i') \in R$ from Lemma 6.38. With $n_i \in dom(\text{obs})$ and using rule (5.10) of Definition 5.7 we see that $(\sigma, i) \in C$ holds as required.

In the latter case we obtain that $(\sigma, \iota) \in R$ and $(\sigma', i') \in R$ by Lemma 6.38. Using rule (5.8) of Definition 5.7 we then have that $(\sigma, i) \in R$ and again with $n_i \in dom(\text{obs})$ and using rule (5.10) of Definition 5.7 we also obtain $(\sigma, i) \in C$ as required. ☐

7 Applications for Information Flow Control

In this chapter we seek to demonstrate the versatility of our previous development by describing some possible applications for information flow control purposes. We describe different approaches how our semantic characterisation can be abstracted to obtain less precise but more tractable properties. Our approaches are axiomatic and provide different interfaces for the implementation of concrete analyses, such that precision can be traded for efficiency and based on the capabilities of a possibly underlying safety analysis as well as the desired degree of integration, a more or less simplified interface might be chosen. Firstly, in Section 7.1 we describe a crude abstraction of our criterion based only on the control flow and data abstractions themselves and thereby obtain what is known as the program dependence graph. In Section 7.2 we then sketch a fixed point–based approach, which we develop stepwise by refining abstract soundness properties derived from our characterisation for increasingly structured abstractions of program semantics. Finally, in Section 7.3 we give a regular approximation of critical executions, which we then utilise in two ways. Firstly, we describe in Section 7.3.3 how this regular abstraction can be folded into the original program to obtain a new program together with a safety property, such that it suffices to verify the safety of the derived program to show the security of the original program. This allows one to utilise off the shelf safety analyses for security analyses. Secondly, the regular approximation forms the basis for our prototypical implementation within an existing program analysis framework for safety analyses, which we present in Section 7.3.4. Throughout this chapter we again fix a program $(\Sigma, N, E, \mathfrak{st}, \mathfrak{te}, [\![.]\!], \mathrm{def}, \mathrm{use}, L, \mathrm{obs})$, admissible in the sense of Definition 4.2.

7.1 Relation to Program Dependence Graphs

We mentioned earlier that our approach is inspired by how program dependence graphs (PDGs) track information flows through control and data dependencies. In this section we want to make this connection more concrete by providing a definition of PDGs for our program model based on classical definitions from the literature and proving connecting theorems. PDGs were introduced by Ferrante et al. [16] as a tool for program optimisation and later exploited for information flow control [25].

We take the definition given by Ferrante et al. [17] as reference. They use control flow graphs which are similar to our control flow abstraction $(N, E, \mathfrak{st}, \mathfrak{te})$ only with the additional restrictions that N must be finite and that any node (control location) might have at most two successors and must be reachable from the initial node. They also utilise the same notion of (strict) post dominance. Based on this their definition of control dependence is as follows.

Definition 7.1 (Definition 3 in [17])**.** Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X if and only if (1) there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and (2) X is not post-dominated by Y.

While phrased slightly differently, this definition directly corresponds to the existence of a path in the control flow abstraction with an immediate control dependence between indices reaching the corresponding nodes. We formalise this in the following lemma.

Lemma 7.1. *A node n is control dependent upon a node m in the sense of Definition 7.1 if and only if there exists a path π and index i such that $n_0 = m$, $n_i = n$ and $0 \xrightarrow{icd}_\pi i$.*

Proof. The "if" direction follows directly from our previous results. We have $\neg n \xrightarrow{pd} m$ from our definition of control dependence and with Lemma 6.22 it follows that $\forall k \in (0, i)\colon n \xrightarrow{pd} n_k$, whereby π is the required path in Definition 7.1.

For the "only if" direction we let π be the path from Definition 7.1 and i be the first instance with $n_i = n$. As we have for any $j \in (0, i)$ that $n_i \xrightarrow{pd} n_j$, it cannot be the case that any such n_j post dominates m, as the transitivity of the post dominator relation (Lemma 6.4) would otherwise yield $n \xrightarrow{pd} m$. It must therefore be the case that $0 \xrightarrow{cd}_\pi i$ and it must moreover already be the case that $0 \xrightarrow{icd}_\pi i$, as if there was any $j \in (0, i)$ with $j \xrightarrow{cd}_\pi i$ this would contradict $n_i \xrightarrow{pd} n_j$. □

Besides control dependencies, the program dependence graph contains data dependencies, which Ferrante et al. [17] define as those being obtained by computing reaching definitions and connecting the definitions of a variable to its uses accordingly. Note, that Ferrante et al. [17] actually include additional nodes obtained by unfolding expressions and connecting those. This however makes no difference for information flow purposes, as one is only interested in the reachability of certain nodes in the PDG, for which the existence of these additional intermediate nodes is irrelevant. If we assume that the reaching definition analysis propagates definitions along program paths, this corresponds directly to the existence of a path connecting the nodes in question and exhibiting one of our data dependencies for some variable, which we therefore use to formally define our notion of a PDG for our program model.

Definition 7.2. The PDG is defined as the graph (N, E_{PDG}) where $(m, n) \in E_{PDG}$ if and only if there exists a path π and indices i, j such that $m = n_i$, $n = n_j$ and either $i \xrightarrow{icd}_\pi j$ or there exists $v \in \mathrm{Var}$ such that $i \xrightarrow{dd_v} j$.

PDGs are exploited for information flow control through slicing. The forward slice of a node in a program is meant to be a safe approximation of the set of nodes that can be influenced by the former and can be obtained from the PDG as the set of nodes that are reachable from the node in question. In our setting we assumed that the confidential information only enters the program through the initial node and the PDG-based approach therefore deems the program secure, if the forward slice of the initial node, $R_{PDG} = \{n \mid (\mathfrak{st}, n) \in E^*_{PDG}\}$ does not contain any observable nodes. As the PDG can be understood as a path-insensitive over-approximation of R, we directly obtain the soundness of the PDG for information flow control.

Corollary 7.2. *If $R_{PDG} \cap \mathrm{dom}(\mathrm{obs}) = \emptyset$ holds, then the program is secure.*

Proof. By induction over the definition of R we first show that for all σ and i:

$$(\sigma, i) \in R \Rightarrow n_i \in R_{PDG} \tag{7.1}$$

The base case for $(\sigma, 0) \in R$ follows directly, as $n_0 = \mathfrak{st}$ is trivially reachable from $\mathfrak{st}$. The cases for data and control dependencies follow directly from the induction hypothesis using the definition of the PDG and the fact that any control dependency can always be decomposed into a sequence of immediate control dependencies by utilising Lemma 6.10.

The case for data control dependencies follows as indices with matching control slices reach the same locations and if $i \xrightarrow{dcd_v^{\sigma',i'}} j$ there are k', j' such that $i =^{\mathrm{cs}}_{\sigma,\sigma'} i' \xrightarrow{cd}_{\sigma'} k' \xrightarrow{dd_v}_{\sigma'} j' =^{\mathrm{cs}}_{\sigma',\sigma} j$ and thereby with the induction hypothesis $n_j = n'_{j'} \in R_{PDG}$.

With (7.1) we then obtain from $R_{PDG} \cap \mathrm{dom}(\mathrm{obs}) = \emptyset$ that $C = \emptyset$ and thereby the program is secure due to Corollary 5.5. □

The fact that a sound PDG can be directly defined through the dependencies we used in our approach serves not only to highlight the connection between the approaches, it also gives us a lower bound for precision of many analyses derived from our approach. Any reasonable analysis which is capable of tracking flows at least on a control location level should not deem any program insecure that can be verified as secure by the PDG-based approach. We explicitly prove this property for our regular approach from Section 7.3 in Lemma 7.11. Moreover one might envision how this direct connection could be exploited to develop optimised variants of the approaches presented in the following sections by performing a PDG-based analysis first and then guiding and pruning the more expensive propagation of the more precise approaches according to the critical paths in the PDG.

7.2 Fixed Point Abstraction

In this section we perform an abstraction of our criterion for single critical reaching executions from Definition 5.7 using a fixed point–based approach. As our goal is to demonstrate the usefulness of our approach within the wide design space that is given here as broadly as possible, we do this in a stepwise manner. In Section 7.2.1 we begin with a very broad definition of a fixed point–based information flow abstraction that contains requirements to reduce the soundness of that definition to our characterisation. In a subsequent step in Section 7.2.2 we instantiate this definition for a class of more concrete abstractions, which propagate abstract states between abstract control locations through abstract dependency transformers that reflect the requirements from our semantic characterisations for data and control dependencies. In Section 7.2.3 we then refine this instantiation further to obtain a more efficient approach and finally demonstrate how this approach can verify the security of a concrete example program using an appropriate abstraction of the program semantics.

7.2.1 Top Level Information Flow Abstraction

In the following definition we capture the basic structure of what we mean by a fixed point–based abstraction of our criterion of critical reaching executions C, respectively reaching executions R as defined in Definition 5.7. Such an abstraction defines a domain of abstract flow facts that represent sets of points within executions, which is sets of state index pairs as we use them in our definition of critical reaching executions. This representation is formalised through an abstraction function that maps state index pairs to abstract flow facts. We then assume that the abstract flow facts form a partial order and based on this any abstract flow fact that is greater or equal than the image of a state index pair under the abstraction map is interpreted as an abstraction representing that state index pair. The dependencies between state index pairs, which are used in the inductive definition of the reaching executions R, are represented by an endofunction on abstract flow facts, which we call an abstract dependency transformer, that yields for any abstract flow fact that represents the source of a dependency an abstract flow fact, which represents the sink of the dependency. In that way a pre-fixed point of this transformer which abstracts all initial state index pairs must be a valid abstraction for all critical state index pairs from C. Thereby, if there exists a suitable pre-fixed point which is not an abstraction of any state index pair from C, then it follows that C is empty and the program is secure. At this point we do not make any assumptions on how the analysis would check for the existence of such a pre-fixed point, but under suitable conditions, e.g. if the underlying domain is a chain complete partial order of finite height and the abstract dependency transformer is continuous, it can be computed by iterative application of the abstract dependency transformer itself.

Definition 7.3. A *single execution information flow abstraction* is a tuple $((D, \sqsubseteq), \alpha, F)$ where $(D, \sqsubseteq)$ is a partial order of *abstract flow facts*, $\alpha\colon \Sigma \times \mathbb{N} \to D$ is an *abstraction function* and $F\colon D \to D$ is an *abstract dependency transformer* which fulfils the following soundness conditions for all σ, i, j, d, v, σ', i':

$$(\sigma, i) \in R \land i \xrightarrow{dd_v}_\sigma j \ \land \alpha(\sigma, i) \sqsubseteq d \quad \Rightarrow \alpha(\sigma, j) \sqsubseteq F(d) \tag{7.2}$$

$$(\sigma, i) \in R \land i \xrightarrow{cd}_\sigma j \ \land \alpha(\sigma, i) \sqsubseteq d \quad \Rightarrow \alpha(\sigma, j) \sqsubseteq F(d) \tag{7.3}$$

$$\begin{aligned}(\sigma, i) \in R \land i \xrightarrow{dcd_v^{\sigma',i'}}_\sigma j \land \alpha(\sigma, i) \sqsubseteq d \land \\ (\sigma', i') \in R \ \land \alpha(\sigma', i') \sqsubseteq d \quad \Rightarrow \alpha(\sigma, j) \sqsubseteq F(d)\end{aligned} \tag{7.4}$$

A *solution* for a single execution information flow abstraction is a flow fact $d \in D$ which is a pre-fixed point of F (that is $F(d) \sqsubseteq d$) such that for all $\sigma \in \Sigma$ it holds that $\alpha(\sigma, 0) \sqsubseteq d$. A solution d is called *safe* if there exists no $(\sigma, i) \in C$ with $\alpha(\sigma, i) \sqsubseteq d$.

As sketched in the motivation for this definition, the assumptions are such that any suitable pre-fixed point, which we call a solution, is a valid abstraction of any state index pair in C. Any solution is actually a valid abstraction for any state index pair in R as formalised in Lemma 7.3, which follows easily as conditions 7.2 to 7.4 directly correspond to the inductive definition of R. The critical state index pairs in C are just those elements of R which reached observable locations, such that we can conclude that the program must be secure if a safe solution exists because C must be empty. This is formalised in Lemma 7.4.

Lemma 7.3. *For any solution $d \in D$ of a single execution information flow abstraction $((D, \sqsubseteq), \alpha, F)$ it holds that*

$$\forall (\sigma, i) \in R\colon \alpha(\sigma, i) \sqsubseteq d.$$

Proof by induction over the definition of R. In the first case (5.6) of Definition 5.7 we have $i = 0$ and as d is a solution we obtain that $\alpha(\sigma, 0) \sqsubseteq d$ as required.

In the second case (5.7) where $(\sigma, j) \in R$ due to $(\sigma, i) \in R$ and $i \xrightarrow{dd_v}_\sigma j$ we have by the induction hypothesis $\alpha(\sigma, i) \sqsubseteq d$ and thereby with (7.2) that $\alpha(\sigma, j) \sqsubseteq F(d) \sqsubseteq d$. The third case (5.8) where $(\sigma, j) \in R$ due to $(\sigma, i) \in R$ and $i \xrightarrow{cd}_\sigma j$ works completely analogously using (7.3).

In the fourth case (5.9) where $(\sigma, j) \in R$ due to $(\sigma, i) \in R$ and $(\sigma', i') \in R$ with $i \xrightarrow{dcd_v^{\sigma',i'}}_\sigma j$ we again obtain via the induction hypothesis $\alpha(\sigma, i) \sqsubseteq d$ and $\alpha(\sigma', i') \sqsubseteq d$, whereby with (7.4) it holds that $\alpha(\sigma, j) \sqsubseteq F(d)$ and with $F(d) \sqsubseteq d$ it follows that $\alpha(\sigma, j) \sqsubseteq d$ as required. □

Lemma 7.4. *If a single execution information flow abstraction of a program has a safe solution, then the program is secure.*

Proof. Assume a safe solution d exists and that the program is not secure. By Corollary 5.5 we obtain the existence of $(\sigma, i) \in C \subseteq R$. With Lemma 7.3 we obtain $\alpha(\sigma, i) \sqsubseteq d$, which is a contradiction to d being safe. □

7.2.2 Abstract Location Based Instantiation

While the definition in the previous section does not assume any structure on the domain of flow facts besides forming a partial order, we will now consider a more structured class of abstractions. Based on this additional structure we will then make more fine-grained assumptions and definitions which we use to instantiate our general definition. A common approach which we utilise here is to use maps from abstract locations to abstract states as flow facts.

We do not assume any structure on the abstract locations themselves and only require an abstraction function which maps points in executions, that is state index pairs as above, to abstract locations, which are mostly meant to represent the location reached by the execution. The abstraction is free to store more information in these locations, however our intention is to utilise those locations to store abstractions for matching indices in executions starting from low equivalent input states like they are used in data control dependencies. We therefore require that matching indices in executions starting from low equivalent input states are abstracted to the same abstract location.

Definition 7.4. A *location abstraction* is a map ρ from $\Sigma \times \mathbb{N}$ to an arbitrary set of so called *abstract locations* X that factors over the reached control slice and the values of low variables, which is $\exists g\colon \forall \sigma, i\colon \rho(\sigma, i) = g(\sigma{\restriction}_L, \mathrm{cs}^\sigma_i)$.

While an abstract location might include further information about the low part of the input state, we utilise it to be a representative for a set of control slices. One might notice that the control slice during an execution behaves somewhat similar to a call stack during the execution of an procedural program, with the slight difference that for control slices all copies of a loop head are *popped* when a loop terminates. Call strings are one technique that is utilised in the analysis of procedural programs to represent subsets from an infinite set of call stacks by common suffixes or other subwords of bounded length. This approach lends itself straightforwardly to be applied in this setting and Example 7.1 can be seen as a special case of this where we utilise *call strings* of length one, which we will use in later examples.

Example 7.1. The *canonical location abstraction*, especially if the set of control locations N is finite, is given by using $X = N$ and defining $\rho(\sigma, i) = n_i$. As cs^σ_i always contains n_i as its last element, ρ is a valid location abstraction.

The flow facts in our domain will map abstract locations to abstract states. An abstract state is intended to be a representative for a set of states that might be observed at a point which is mapped to a given abstract location but it might contain any information about the initial state or point within the execution. Formally we assume an abstraction function that maps state index pairs to abstract states which we require to be non-bottom elements of a complete lattice. We make the assumption that the abstract states form a complete lattice and require that the abstraction function does not map concrete states to the bottom element in order to simplify the following definitions where we will utilise the bottom element to denote unreached locations and use least upper bounds in places where any upper bound would suffice for soundness purposes.

Definition 7.5. A *state abstraction* is a map $\alpha^\Sigma \colon \Sigma \times \mathbb{N} \to \Sigma^\#$ for a complete lattice $(\Sigma^\#, \sqsubseteq)$ such that $\bot \notin \operatorname{ran}(\alpha^\Sigma)$.

From the world of abstraction-based safety analyses there is an abundance of state abstractions like various numerical abstract domains and others to choose from. Note that many will either be relational domains and come in the form of abstractions of the reached state itself $\bar{\alpha} \colon \Sigma \to \Sigma^\#$, which would simply be lifted to state index pairs via $\alpha^\Sigma(\sigma, i) = \bar{\alpha}(\sigma_i)$ or non-relational abstractions of values stored in variables, where one might have a collection of potentially different abstractions $\alpha_v \colon \mathrm{Val} \to \Sigma^\#_v$ one for each variable $v \in \mathrm{Var}$ and one defines a state abstraction pointwise for each variable by $\alpha^\Sigma(\sigma, i) = v \mapsto \alpha_v(\sigma_i(v))$. For our examples we will utilise a so called predicate abstraction where abstract states are formal predicates and represent the set of states in which they hold based on a suitable interpretation. We give a definition for this in Example 7.2.

Example 7.2. A predicate abstraction is defined by fixing a set of predicates Pred together with an interpretation $\models \colon \Sigma \times \mathit{Pred} \to \mathbb{B}$ which fulfils that Pred forms a complete lattice with respect to $p \sqsubseteq q \Leftrightarrow \forall \sigma \in \Sigma \colon \sigma \models p \Rightarrow \sigma \models q$ and guarantees for all σ that $\sigma \not\models \bot$ as well as that the mapping $p \mapsto \sigma \models p$ preserves all meets, that is $\forall P \subseteq \mathit{Pred} \colon (\sigma \models \bigsqcap P) \Leftrightarrow (\forall p \in P \colon \sigma \models p)$. We then define the state abstraction $\alpha^\Sigma \colon \Sigma \times \mathbb{N} \to \mathit{Pred}$ by $\alpha^\Sigma(\sigma, i) = \bigsqcap\{p \in \mathit{Pred} \mid \sigma_i \models p\}$. We observe that α^Σ is indeed a valid state abstraction. For this we only need to verify that $\alpha^\Sigma(\sigma, i) \neq \bot$. Due to the fact that $p \mapsto \sigma_i \models p$ preserves all meets, we have $\sigma_i \models \alpha^\Sigma(\sigma, i)$, as $\sigma_i \models \alpha^\Sigma(\sigma, i) \Leftrightarrow (\forall p \in \{p \in \mathit{Pred} \mid \sigma_i \models p\} \colon \sigma_i \models p)$. We have by assumption that $\sigma_i \not\models \bot$ whereby $\alpha^{\Sigma}(\sigma, i) \neq \bot$ holds as required.

Note that one can define equally expressive notions of predicate abstractions as above which do not require the strict assumptions made on the set of predicates but this definition allows for simpler handling without having to deal with potentially ambiguous representations for equivalent predicates and the like.

We use Example 4.2 as a running example in this section and in Example 7.3 define an appropriate predicate abstraction which we will utilise for its analysis. This is a minimised version only containing the strictly necessary predicates to verify the program's security. In practice including guards appearing in the program as predicates is a common technique and we might envision for instance that a user who has basic knowledge about the program could configure the analysis with additional potentially beneficial predicates, such as the ones about evenness used in our example.

Example 7.3. For the verification of Example 4.2 we utilise a predicate abstraction with $Pred = \{\top, i < 2u, even(i), odd(i), i < 2u \wedge even(i), i < 2u \wedge odd(i), \bot\}$, which, with the canonical interpretation $\models$, forms a complete lattice that is depicted in Figure 7.1. One can verify that for all σ the mapping $p \mapsto \sigma \models p$ preserves all meets and that $Pred$ with $\models$ defines a valid state abstraction through Example 7.2.

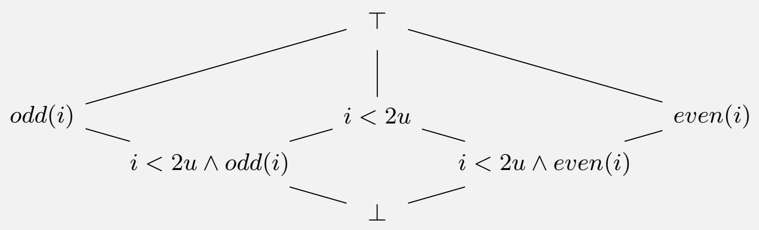


Figure 7.1: Complete lattice for the verification of Example 4.2.

As mentioned above, we will utilise maps from abstract locations to abstract states as flow facts for instantiating Definition 7.3. The corresponding abstraction based on given location and state abstractions is formally defined in Definition 7.6. The flow fact for a given state index pair under this abstraction maps the abstract location given by the location abstraction to the abstract state given by the state abstraction and all other abstract locations to the abstract bottom state.

Definition 7.6. Given a location abstraction $\rho\colon \Sigma \times \mathbb{N} \to X$ and state abstraction $\alpha^{\Sigma}\colon \Sigma \times \mathbb{N} \to \Sigma^{\#}$ the *induced domain of flow facts* is $D = X \to \Sigma^{\#}$ where $\sqsubseteq$ is lifted to D pointwise through $d \sqsubseteq d' \Leftrightarrow \forall x \in X\colon d(x) \sqsubseteq d'(x)$ and the *induced abstraction* $\alpha\colon \Sigma \times \mathbb{N} \to D$ is defined by

$$\alpha(\sigma, i) = x \mapsto \begin{cases} \alpha^{\Sigma}(\sigma, i) & \text{if } x = \rho(\sigma, i), \\ \bot & \text{otherwise.} \end{cases}$$

With the additional structure on flow facts given by these definitions we can define a more direct criterion for pre-fixed points to be (safe) solutions if they utilise the location abstraction from Example 7.1.

Lemma 7.5. *For a single execution information flow abstraction that is using the abstraction obtained from Definition 7.6 based on the location abstraction $\rho(\sigma, i) = n_i$ from Example 7.1 it holds that any pre-fixed point d with $d(\mathfrak{st}) = \top$ is a solution and moreover d is a safe solution if it also holds that $\forall n \in \mathrm{dom}(\mathrm{obs})\colon d(n) = \bot$.*

Proof. Firstly, for all σ it holds that $\rho(\sigma, 0) = n_0 = \mathfrak{st}$ and $\alpha^{\Sigma}(\sigma, 0) \sqsubseteq \top = d(\mathfrak{st})$. Thereby $\alpha(\sigma, 0) \sqsubseteq d$ holds by Definition 7.6 as required. For the second claim we have for all $(\sigma, i) \in C$ that $\rho(\sigma, i) = n_i \in \mathrm{dom}(\mathrm{obs})$ and $\alpha(\sigma, i)(n_i) = \alpha^{\Sigma}(\sigma, i) \sqsupset \bot = d(n_i)$ whereby it follows that d is safe because it cannot be the case that $\alpha(\sigma, i) \sqsubseteq d$. ☐

As it is common in abstract interpretation inspired approaches like this we will finally rely on an abstract version of the semantic step function that allows us to compute abstract successors in order to define the abstract dependency transformer required by Definition 7.3. While our program semantics is deterministic and only yields a single successor location and state for any location state pair, an abstraction will often be non-deterministic and propagate an abstract state from one abstract location to several others. We therefore require a so called semantic abstraction, which for any given abstract source location, state and target location yields an abstract state that is to be propagated to the target location and safely abstracts the program semantics as formalised in Definition 7.7.

Definition 7.7. Given location and state abstractions ρ and α^{Σ}, a *semantic abstraction* is a mapping $f\colon X \times \Sigma^{\#} \times X \to \Sigma^{\#}$ monotone in the second argument, such that

$$\alpha^{\Sigma}(\sigma, i+1) \sqsubseteq f(\rho(\sigma, i), \alpha^{\Sigma}(\sigma, i), \rho(\sigma, i+1)) \tag{7.5}$$

We lift f to an endofunction $f\colon D \to D$ by

$$f(d) = x \mapsto \bigsqcup\bigl\{f(x', d(x'), x) \bigm| x' \in X\bigr\}.$$

With our usage of the bottom state to indicate unreachability, it is desirable that a semantic abstraction is strict in the second argument, that is it always maps the bottom state to itself, but this is not required for soundness. We note that as a semantic abstraction is required to be monotone in the second argument, we can utilise (7.5) to formally define a *best* semantic abstraction which takes the join over $\alpha^\Sigma(\sigma, i+1)$ for all relevant σ and i by:

$$f(x', \sigma^\#, x) = \bigsqcup\{\alpha^\Sigma(\sigma, i+1) \mid \alpha^\Sigma(\sigma, i) \sqsubseteq \sigma^\# \wedge x' = \rho(\sigma, i) \wedge x = \rho(\sigma, i+1)\}$$

This definition is however not well suited for an effective analysis as it will not be computable in general and one will want to provide a suitable alternative. For programs from our command language with simple arithmetic expressions suitable semantic abstractions exist for many common abstract domains and we provide one which we use to verify our running example in Example 7.4.

Example 7.4. We again consider the program from Example 4.2. Based on the location abstraction from Example 7.1 and predicate abstraction from Example 7.3 we define the semantic abstraction $f\colon N \times \mathit{Pred} \times N \to \mathit{Pred}$ as follows. Table 7.1 contains the definition of f for some special location pairs.

Table 7.1: Special cases of the semantic abstraction.

n'	n	$f(n', p, n)$ for $p \neq \bot$
`i = 0`	`x = h`	$\mathit{even}(i)$
`i = 1`	**`while`** `i < 2*u`	$\mathit{odd}(i)$
`while` `i < 2*u`	`i = i + 1`	$p \sqcap i < 2u$
`while` `i < 2*u`	**`print`** `y`	$\begin{cases} \bot & \text{if } p \sqsubseteq i < 2u \\ p & \text{otherwise} \end{cases}$
`i = i + 1`	`z = x`	$\begin{cases} \mathit{odd}(i) & \text{if } p = \mathit{even}(i) \\ \mathit{even}(i) & \text{if } \bot \sqsubset p \sqsubseteq \mathit{odd}(i) \\ i < 2u \wedge \mathit{odd}(i) & \text{if } p = i < 2u \wedge \mathit{even}(i) \\ \top & \text{if } i < 2u \sqsubseteq p \end{cases}$

For all $n, n' \in N$ we define $f(n', \bot, n) = \bot$. For the location pairs not listed in Table 7.1 we have that either $f(n', p, n) = p$ if $(n', n) \in E$ or $f(n', p, n) = \bot$ if $(n', n) \notin E$. It is straightforward to verify through careful case distinction that this definition is indeed monotone in the second argument and that it fulfils (7.5).

With the above setup we define the abstract dependency transformer as required by Definition 7.3. We do this independently for the three kinds of dependencies. For data dependencies we additionally require abstract versions of the data abstractions (def, use) compatible with the given location abstraction. The abstract version of variables read at a location is required to encompass all variables read by any corresponding control location, that is any location reached by a state index pair abstracted to the abstract location. As a single abstract location might represent multiple control locations, which might not write the same variables, we utilise two maps for variables written at abstract locations. One map encompasses for a given abstract location all variables that might be written by any corresponding control location and another which contains only variables that must be written by all corresponding control locations. Based on these we then define for each variable an abstract transformer by propagating according to the semantic abstraction starting from any abstract location where the variable might be written, then iteratively further over all abstract locations where it is not the case that the variable must be written and finally selecting only those where the variable is read. The join over all those transformers for all variables then yields the final transformer for data dependencies as formalised in the following definition.

Definition 7.8. Given a location abstraction $\rho\colon \Sigma \times \mathbb{N} \to X$, a *read write abstraction* is a triple $(\mathrm{use}^{\#}, \mathrm{def}_{\exists}^{\#}, \mathrm{def}_{\forall}^{\#})$ of mappings from X to 2^{Var} such that for all σ, i:

$$\mathrm{use}(n_i) \subseteq \mathrm{use}^{\#}(\rho(\sigma, i)) \tag{7.6}$$

$$\mathrm{def}(n_i) \subseteq \mathrm{def}_{\exists}^{\#}(\rho(\sigma, i)) \tag{7.7}$$

$$\mathrm{def}_{\forall}^{\#}(\rho(\sigma, i)) \subseteq \mathrm{def}(n_i) \tag{7.8}$$

Based on these we define, when additionally given a state abstraction α^{Σ} and semantic abstraction $f\colon X \times \Sigma^{\#} \times X \to \Sigma^{\#}$, the *abstract read write transformers* as the endofunctions use_v, def_v, $\mathrm{def}_{\neg v}$ for $v \in \mathrm{Var}$ on D such that for all $d \in D, x \in X$:

$$\mathrm{def}_v(d)(x) = \bigsqcup \bigl\{ f(x', d(x'), x) \bigm| x' \in X, v \in \mathrm{def}_{\exists}^{\#}(x') \bigr\} \tag{7.9}$$

$$\mathrm{def}_{\neg v}(d)(x) = \bigsqcup \big\{ f(x', d(x'), x) \;\big|\; x' \in X, v \notin \mathrm{def}^{\#}_{\forall}(x') \big\} \tag{7.10}$$

$$\mathrm{use}_v(d)(x) = \begin{cases} d(x) & \text{if } v \in \mathrm{use}^{\#}(x) \\ \bot & \text{else} \end{cases} \tag{7.11}$$

With these we define the *abstract data dependency transformers* $\mathrm{dd}^{\#}_v$ for $v \in \mathrm{Var}$ and $\mathrm{dd}^{\#}$ on D through:

$$\mathrm{dd}^{\#}_v(d) = \mathrm{use}_v(\mathrm{def}^{*}_{\neg v}(\mathrm{def}_v(d))) \tag{7.12}$$

$$\mathrm{dd}^{\#}(d) = \bigsqcup \big\{ \mathrm{dd}^{\#}_v(d) \;\big|\; v \in \mathrm{Var} \big\} \tag{7.13}$$

Recall that for any endofunction $g\colon D \to D$ on a complete lattice we denote by $g^{*}\colon D \to D$ the endofunction $d \mapsto \bigsqcup\{g^n(d) \mid n \in \mathbb{N}\}$. We note that for correctness we only require any upper bound here, which for instance for monotone g and if D has no infinite strictly ascending chains can be obtained through the Kleene iteration $d_0 = d$, $d_{i+1} = d_i \sqcup g(d_i)$. We will however perform further approximations to remove those closures in Definition 7.12 where we then exploit the use of the least upper bound at this place by proving that Definition 7.12 is a sound over-approximation of this definition. For our examples a read write abstraction can be trivially obtained from the program model as we note in Example 7.5.

Example 7.5. For the location abstraction from Example 7.1 where $\rho(\sigma, i) = n_i$ we can directly define $\mathrm{use}^{\#} = \mathrm{use}$ and $\mathrm{def}^{\#}_{\exists} = \mathrm{def}^{\#}_{\forall} = \mathrm{def}$.

We now define the abstract dependency transformer for control dependencies. Similar to how the definition of control dependence in executions is based upon the post dominance relation on control locations, we base the abstract dependency transformer for control dependencies on an abstraction of control dependence on abstract locations. The definition of control dependence requires that the execution does not visit any post dominating control location. Our abstract version contrariwise maps an abstract location to a set of abstract locations that encompasses all abstractions of control dependent indices. We then define the abstract dependency transformer for control dependencies by iteratively propagating from any abstract location within the set of abstract locations provided by the abstraction of control dependence, which is sound due to the transitivity of the control dependency relation.

Definition 7.9. Given a location abstraction $\rho\colon \Sigma \times \mathbb{N} \to X$, an abstraction of control dependence is a mapping $\mathrm{cds}^{\#}\colon X \to 2^X$ such that for all σ, i, j it holds that

$$i \xrightarrow{cd}_{\sigma} j \Rightarrow \rho(\sigma, j) \in \mathrm{cds}^{\#}(\rho(\sigma, i)) \tag{7.14}$$

With this we define, given a state abstraction that together with ρ induces the domain D, for any $x \in X$ the *control filter* $\mathrm{cds}^{\#}_x\colon D \to D$ by

$$\mathrm{cds}^{\#}_x(d)(y) = \begin{cases} d(y) & \text{if } y \in \mathrm{cds}^{\#}(x) \\ \bot & \text{otherwise.} \end{cases} \tag{7.15}$$

When additionally given a compatible semantic abstraction f we define the *abstract control dependency transformer* by

$$\mathrm{cd}^{\#}_x(d) = (\mathrm{cds}^{\#}_x \circ f)^{+}(d\lfloor_x) \tag{7.16}$$

$$\mathrm{cd}^{\#}(d) = \bigsqcup\{\mathrm{cd}^{\#}_x(d) \mid x \in X\} \tag{7.17}$$

Recall that for $d \in D$ we denote by $d\lfloor_x \in D$ the flow fact that maps x to $d(x)$ and all other abstract locations to the bottom element. Example 7.6 defines the canonical abstraction of control dependence which we use in our running example.

Example 7.6. For the location abstraction from Example 7.1 we can define a valid abstraction of control dependence based on the strict post dominance relation by

$$\mathrm{cds}^{\#}(n) = \{n' \in N \mid \neg\, n' \xrightarrow{pd} n\}.$$

Note that this definition includes unnecessary control locations that lie before the control location or within branches that can be reached after reaching a post dominator. However, for strict semantic abstractions this makes no difference as there happens no further propagation by (7.16) after reaching a first post dominator.

In order to handle data control dependencies, we utilise the transformers already defined for data and control dependencies and exploit the assumption that the abstract bottom state does not represent any concrete state. For any abstract location and variable we first propagate along abstract locations that do not write to the variable to any abstract location reading the variable. We take the abstract states obtained at these reading abstract locations but additionally filter out those whose abstract location is only reached by the abstract bottom state when instead propagating from the original abstract location to its control dependent abstract locations and then to any abstract location data dependent via the fixed variable.

Definition 7.10. Given location, state, read write, control and semantic abstractions we define based on the previous definitions the filter $\triangledown \colon D \times D \to D$ by

$$(d \triangledown c)(x) = \begin{cases} \bot & \text{if } c(x) = \bot \\ d(x) & \text{otherwise} \end{cases} \tag{7.18}$$

as well as the *abstract data control dependency transformer* by

$$\mathrm{dcd}^{\#}(d) = \bigsqcup \left\{ \mathrm{use}_v(\mathrm{def}^{+}_{\neg v}(d\lfloor_x)) \triangledown \mathrm{dd}^{\#}_v(\mathrm{cd}^{\#}_x(d)) \;\middle|\; x \in X, v \in \mathrm{Var} \right\} \tag{7.19}$$

Finally, we combine the three transformers defined above to obtain the desired abstract dependency transformer for our single execution information flow abstraction, which we prove to be valid under the assumptions made in the above definitions in Lemma 7.6.

Definition 7.11. Given a location abstraction $\rho \colon \Sigma \times \mathbb{N} \to X$, a state abstraction $\alpha^{\Sigma} \colon \Sigma \times \mathbb{N} \to \Sigma^{\#}$, a semantic abstraction $f \colon X \times \Sigma^{\#} \times X \to \Sigma^{\#}$, a read write abstraction $\mathrm{use}^{\#}, \mathrm{def}^{\#}_{\exists}, \mathrm{def}^{\#}_{\forall} \colon X \to \mathrm{Var}$ and an abstraction of control dependence $\mathrm{cds}^{\#} \colon X \to 2^X$, we define based on the above definitions the *induced abstract dependency transformer* on $D = X \to \Sigma^{\#}$ by

$$F(d) = \mathrm{dd}^{\#}(d) \sqcup \mathrm{cd}^{\#}(d) \sqcup \mathrm{dcd}^{\#}(d). \tag{7.20}$$

Lemma 7.6. $((D, \sqsubseteq), \alpha, F)$ *as defined by Definition 7.11 forms a valid single execution information flow abstraction in the sense of Definition 7.3.*

Proof. The assumption that $(D, \sqsubseteq)$ forms a partial order follows from the fact that it is a complete lattice, which follows from $(\Sigma^{\#}, \sqsubseteq)$ being a complete lattice.

In order to show that (7.2) to (7.4) hold we first fix σ, $i < j$ and d such that $\alpha(\sigma, i) \sqsubseteq d$ and let $(x_k)_{i \leq k \leq j} = (\rho(\sigma, k))_{i \leq k \leq j}$ and $(\sigma^{\#}_k)_{i \leq k \leq j} = (\alpha^{\Sigma}(\sigma, i))_{i \leq k \leq j}$.

For (7.2) we additionally fix v and presume that $i \xrightarrow{dd_v}_{\sigma} j$. We have to show that $\alpha(\sigma, j) \sqsubseteq F(d)$, which means showing that $\sigma^{\#}_j \sqsubseteq F(d)(x_j)$. As $i \xrightarrow{dd_v}_{\sigma} j$, we have by (7.7) that $v \in \mathrm{def}^{\#}_{\exists}(x_i)$. With this we obtain from (7.9) that $(*)$: $\mathrm{def}_v(d)(x_{i+1}) \sqsupseteq f(x_i, d(x_i), x_{i+1})$. From $\alpha(\sigma, i) \sqsubseteq d$ we have that $d(x_i) \sqsupseteq \sigma^{\#}_i$ and with $(*)$ we obtain, as f is monotone in the second argument, that

$$\mathrm{def}_v(d)(x_{i+1}) \sqsupseteq f(x_i, \sigma^{\#}_i, x_{i+1}) \overset{(7.5)}{\sqsupseteq} \sigma^{\#}_{i+1}.$$

As $v \notin \mathrm{def}(n_k) \supseteq \mathrm{def}^{\#}_{\forall}(x_k)$ for $i < k < j$ we obtain inductively with the same argument as above, using (7.10) instead of (7.9), that $\mathrm{def}^*_{\neg v}(\mathrm{def}_v(d))(x_k) \sqsupseteq \sigma^{\#}_k$ for $i < k \leq j$ by using $\mathrm{def}^*_{\neg v}(\mathrm{def}_v(d)) \sqsupseteq \mathrm{def}_v(d)$ in the base case and $\mathrm{def}^*_{\neg v}(\mathrm{def}_v(d)) \sqsupseteq \mathrm{def}_{\neg v}(\mathrm{def}^*_{\neg v}(\mathrm{def}_v(d)))$ in the inductive case. Thereby, as $v \in \mathrm{use}(n_j) \subseteq \mathrm{use}^{\#}(x_j)$, we obtain as required

$$\sigma^{\#}_j \sqsubseteq \mathrm{def}^*_{\neg v}(\mathrm{def}_v(d))(x_j) \sqsubseteq \mathrm{use}_v(\mathrm{def}^*_{\neg v}(\mathrm{def}_v(d)))(x_j) = \mathrm{dd}^{\#}_v(d)(x_j) \sqsubseteq F(d)(x_j).$$

For (7.3) we additionally assume $i \xrightarrow{cd}_{\sigma} j$. Again we have to verify $\sigma^{\#}_j \sqsubseteq F(d)(x_j)$. By Lemma 6.10 we have $i \xrightarrow{cd}_{\sigma} k$ for k with $i < k \leq j$. For these k we then obtain, from (7.5), (7.14) and (7.15) using the monotonicity of both f and $\mathrm{cds}^{\#}_{x_i}$ with $\bot \sqsubset \sigma^{\#}_k$, that

$$\begin{aligned} \sigma^{\#}_{k-1} \sqsubseteq \hat{d}(x_{k-1}) &\Rightarrow \sigma^{\#}_k \sqsubseteq f(\hat{d})(x_k), \\ \sigma^{\#}_k \sqsubseteq \hat{d}(x_k) \quad &\Rightarrow \sigma^{\#}_k \sqsubseteq \mathrm{cds}^{\#}_{x_i}(\hat{d})(x_k) \text{ and} \\ \sigma^{\#}_{k-1} \sqsubseteq \hat{d}(x_{k-1}) &\Rightarrow \sigma^{\#}_k \sqsubseteq (\mathrm{cds}^{\#}_{x_i} \circ f)(\hat{d})(x_k). \end{aligned}$$

Together with $\sigma^{\#}_i(x_i) \sqsubseteq (d\lfloor_{x_i})(x_i)$ we obtain by induction for any k with $i < k \leq j$ that

$$\sigma^{\#}_k \sqsubseteq (\mathrm{cds}^{\#}_{x_i} \circ f)^+(d\lfloor_{x_i})(x_k).$$

We therefore have $\sigma^{\#}_j \sqsubseteq \mathrm{cd}^{\#}(d)(x_j) \sqsubseteq F(d)(x_j)$ as required.

For (7.4) we additionally assume that $i \xrightarrow{dcd_v^{\sigma',i'}}_{\sigma} j$ with $(\sigma', i') \in R$ and $\alpha(\sigma', i') \sqsubseteq d$. We obtain k' and j' such that $i' \xrightarrow{cd}_{\sigma'} k' \xrightarrow{dd_v}_{\sigma'} j' =^{\mathrm{cs}}_{\sigma',\sigma} j$. As $\sigma =_L \sigma'$ and $i =^{\mathrm{cs}}_{\sigma,\sigma'} i'$ we have $\rho(\sigma', i') = x_i$ as well as $\rho(\sigma', j') = x_j$. From this we obtain from the previous cases that $\bot \sqsubset \alpha^{\Sigma}(\sigma', j') \sqsubseteq \mathrm{dd}^{\#}_v(\mathrm{cd}^{\#}_{x_i}(d))(x_j)$. We have $v \notin \mathrm{def}(n_k) \supseteq \mathrm{def}^{\#}_{\forall}(x_k)$ for $i < k < j$ as well as $v \in \mathrm{use}(n_k) \subseteq \mathrm{use}^{\#}(x_k)$ and obtain in the same manner as in the first case that $\sigma^{\#}_j \sqsubseteq \mathrm{use}_v(\mathrm{def}^+_{\neg v}(d\lfloor_{x_i}))(x_j)$. With Definition 7.10 we finally obtain that $\sigma^{\#}_j \sqsubseteq (\mathrm{use}_v(\mathrm{def}^+_{\neg v}(d\lfloor_{x_i}))\triangledown \mathrm{dd}^{\#}_v(\mathrm{cd}^{\#}_{x_i}(d)))(x_j) \sqsubseteq \mathrm{dcd}^{\#}(d)(x_j) \sqsubseteq F(d)(x_j)$ as required. $\square$

7.2.3 Optimising the Instantiation for Efficiency

The instantiation in the previous section uses transitive closures to directly capture the propagation of flow facts along dependencies. These closures or at least safe upper bounds, should usually be computable under similar assumptions as are generally required for abstract domains and transformers to be effective. For appropriate instantiations we therefore already have a computable analysis. Yet we were quite liberal in utilising those closures and having to repeatedly recompute the corresponding nested pre-fixed points can lead to rather expensive analyses. We will now unfold those closures into the main transformer itself where we are free to merge several of them to obtain a cheaper analysis that propagates in a stepwise manner directly along program paths.

In order to achieve this goal we extend the domain X of flow facts with different copies for the various kinds of dependencies. Another approach would be to instead enrich the abstract states associated with each location with the required information, which with the transformation used below would yield an even cheaper analysis at the cost of loosing further precision. After this transformation the propagation by the new transformer will not work in the same big step manner along dependencies. Hence it is not a direct instance of the above definition but it will be defined in such a way that any pre-fixed point of the extended version restricted to the original domain X will be a pre-fixed point of the abstract dependency transformer from Definition 7.11. Depending on the distributivity of the underlying domain and continuity of the transformers, different choices on where to split and merge the domain and dependencies can lead to cruder or finer over-approximations. Therefore in practise it would be advisable to make an informed decision based on the problem and tools at hand. Here we make choices that allow us to handle our examples and illustrate the propagation in a reasonably concise way. For data dependencies we split the domain based on the variable propagating the dependency. To this end we extend the domain by $X \times \mathrm{Var}$, where the second component represents the variable of the data dependency and the first component the abstract location reached after writing to the variable without overwriting the information. For control dependencies we proceed in the same manner and split the domain based on the abstract location of where a branch was entered and therefore extend the domain further by $X \times X$. We do not utilise additional copies for data control dependencies but reuse those for data and control dependencies.

Definition 7.12. Given location and state abstractions ρ and α^{Σ} with induced abstraction α, a semantic abstraction $f\colon X \times \Sigma^{\#} \times X \to \Sigma^{\#}$ which is strict in the second argument (that is $\forall x, y\colon f(x, \bot, y) = \bot$), a read write abstraction $(\mathrm{use}^{\#}, \mathrm{def}^{\#}_{\exists}, \mathrm{def}^{\#}_{\forall})$ and an abstraction of control dependence $\mathrm{cds}^{\#}$, we define $\hat{D} = X \cup (X \times \mathrm{Var}) \cup (X \times X) \to \Sigma^{\#}$ with the pointwise order $\sqsubseteq$ induced by $(\Sigma^{\#}, \sqsubseteq)$ and define the induced *small step abstract information flow transformer* $\hat{F}\colon \hat{D} \to \hat{D}$ by

$$\begin{aligned}
\hat{F}(\hat{d})(x) = \bigsqcup\Big(&\big\{\hat{d}(x,v) \bigm| v \in \mathrm{use}^{\#}(x)\big\} \cup \\
&\big\{\hat{d}(x,y) \bigm| y \in X\big\}\Big) && (7.21)\\
\hat{F}(\hat{d})(x,v) = \bigsqcup\Big(&\big\{f(x', \hat{d}(x'), x) \bigm| v \in \mathrm{def}^{\#}_{\exists}(x')\big\} \cup \\
&\big\{\hat{d}(x) \bigm| \exists y\colon v \in \mathrm{def}^{\#}_{\exists}(y) \wedge \hat{d}(y,x) \sqsupset \bot\big\} \cup \\
&\big\{f(x', \hat{d}(x',v), x) \bigm| v \notin \mathrm{def}^{\#}_{\forall}(x')\big\}\Big) && (7.22)\\
\hat{F}(\hat{d})(x,y) = \bigsqcup\Big(&\big\{f(x', \hat{d}(x',y), x) \bigm| x' \in X \wedge x \in \mathrm{cds}^{\#}(y)\big\} \cup \\
&\big\{\hat{d}(y) \bigm| x = y\big\}\Big) && (7.23)
\end{aligned}$$

As for single execution information flow abstractions we call $\hat{d} \in \hat{D}$ a *solution* if it is a pre-fixed point of $\hat{F}$ and $\alpha(\sigma, 0) \sqsubseteq \hat{d}{\restriction_X}$ for all σ. We call it *safe* if there exists no $(\sigma, i) \in C$ with $\alpha(\sigma, i) \sqsubseteq \hat{d}{\restriction_X}$.

The propagation by $\hat{F}$ uses the main copy for the abstractions of the state index pairs of reaching executions from R as is done by F above. Within the main copy there is no direct propagation between abstract locations and all propagation happens through the copies for the individual dependencies. For data dependencies we begin by propagating from the main copy, at any location where a variable is written to its successors within the copy for the variable in question, as is done in the first line of (7.22). Within the copy for a variable we propagate along program paths as long as we cannot guarantee that the variable has been overwritten, which is done in the third line of (7.22). Finally at any location where a variable is read we propagate from the corresponding copy back into the main copy to complete the propagation of the data dependency, which is done in the first part of (7.21). For control dependencies we propagate from the main copy to the copy of the current abstract location, as is done in the second part of (7.23), and propagate there further as long as the abstraction of control dependence deems the reached location control dependent, as is done in the first part of (7.23). From any abstract location reached that way we directly propagate back to the main copy, as is done in the second part of (7.21). For data control dependencies we exploit the fact that the abstract bottom state does not represent any concrete state and whenever in the copy of an abstract location a control dependent definition of a variable is reached by a non-bottom state, we propagate from the controlling abstract location in the main copy to that abstract location in the copy for that variable, which is done in the second line of (7.22). That means that we actually drop the requirement that the data dependency is actually realised in another execution but handle a branching location as if it actually writes any variable that we see modified at a control dependent location in another critical execution. We choose to do so as it is cheap to realise in this setting and will be just as precise in most practical cases, as when there is a controlled write that is not read then either the phantom write added to the branching location will not be read either or whatever keeps the reading location from being reached, after taking the high influenced branch with the write, should itself already be high influenced.

Note that strictness, which we assumed for the semantic abstraction, is a desirable property for semantic abstractions in most cases but here we actually require it as we want solutions of $\hat{F}$ to be solutions of F. The property is necessary as we shortcut the criterion for data control dependencies and a non-strict semantic abstraction might lead to spurious propagation of data control dependencies in F that would not be propagated by $\hat{F}$ which could break the following lemma.

Lemma 7.7. *For $\hat{F}$ as defined in Definition 7.12 and F as defined in Definition 7.11 we have that $\hat{F}(\hat{d}) \sqsubseteq \hat{d} \Rightarrow F(\hat{d}\restriction_X) \sqsubseteq \hat{d}\restriction_X$.*

Proof. Let $\hat{F}(\hat{d}) \sqsubseteq \hat{d}$ hold. Utilising (7.20) it suffices to show that $\hat{d}\restriction_X$ is an upper bound for $\mathrm{dd}^{\#}(\hat{d}\restriction_X)$, $\mathrm{cd}^{\#}(\hat{d}\restriction_X)$ and $\mathrm{dcd}^{\#}(\hat{d}\restriction_X)$.

For the first case we have $\mathrm{def}_v(\hat{d})(x) \sqsubseteq \hat{d}(x, v)$ as the supremum in (7.22) is taken over a greater set than the one in (7.9) in the definition of def_v. Inductively we obtain from the monotonicity of f that $\mathrm{def}^*_{\neg v}(\mathrm{def}_v(\hat{d}\restriction_X))(x) \sqsubseteq \hat{d}(x, v)$, as (7.22) subsumes (7.10). Thereby we obtain $\mathrm{dd}^{\#}(\hat{d}\restriction_X) \sqsubseteq \hat{d}\restriction_X$ as required because all elements not mapped to bottom in (7.11) have a supremum in (7.21).

For the second case we obtain $\hat{d}(x) \sqsubseteq \hat{d}(x, x)$ from (7.23) and thereby $(\hat{d}\restriction_X)\lfloor_x(y) \sqsubseteq \hat{d}(y, x)$ for all x and y. With this we inductively obtain $(\mathrm{cds}^{\#}_x \circ f)^+((\hat{d}\restriction_X)\lfloor_x)(y) \sqsubseteq \hat{d}(y, x)$ by unfolding (7.15), as it holds trivially for $y \notin \mathrm{cds}^{\#}(x)$ and otherwise follows from (7.23) with $\hat{F}(\hat{d}) \sqsubseteq \hat{d}$. Together with $\hat{d}(y, x) \sqsubseteq \hat{d}(y)$, which holds due to (7.21), we obtain $\mathrm{cd}^{\#}(\hat{d}\restriction_X) \sqsubseteq \hat{d}\restriction_X$ as required.

Lastly we show $(\mathrm{use}_v(\mathrm{def}^{+}_{\neg v}((\hat{d}\restriction_X)\lfloor_x))\triangledown \mathrm{dd}^{\#}_v(\mathrm{cd}^{\#}_x(\hat{d}\restriction_X)))(y) \sqsubseteq \hat{d}(y,v)$ for arbitrary x, y and v. This then implies $\mathrm{dcd}^{\#}(\hat{d}\restriction_X) \sqsubseteq \hat{d}\restriction_X$ by (7.21). First consider the case that $\mathrm{dd}^{\#}_v(\mathrm{cd}^{\#}((\hat{d}\restriction_X)\lfloor_x))(y) = \bot$. In that case the left-hand side is bottom and the inequation holds trivially by (7.18). Otherwise we obtain from the strictness of f that $\mathrm{cd}^{\#}_x(\hat{d}\restriction_X)(x') \sqsupset \bot$ for some x' with $v \in \mathrm{def}^{\#}_{\exists}(x')$. From the previous cases we know that $\mathrm{cd}^{\#}_x(\hat{d}\restriction_X)(x') \sqsubseteq \hat{d}(x',x)$ thereby $\hat{d}(x',x) \sqsupset \bot$ and we obtain $\hat{d}(x) \sqsubseteq \hat{d}(x,v)$. Inductively we again obtain $\mathrm{def}^{+}_{\neg v}((\hat{d}\restriction_X)\lfloor_x)(y) \sqsubseteq \hat{d}(y,v)$ and thereby the required

$$(\mathrm{use}_v(\mathrm{def}^{+}_{\neg v}((\hat{d}\restriction_X)\lfloor_x))\triangledown \mathrm{dd}^{\#}_v(\mathrm{cd}^{\#}_x(\hat{d}\restriction_X)))(y) \sqsubseteq \mathrm{use}_v(\mathrm{def}^{+}_{\neg v}((\hat{d}\restriction_X)\lfloor_x))(y) \sqsubseteq \hat{d}(y,v).$$

□

As the definition of solutions of $\hat{F}$ otherwise directly matches that for single execution information flow abstractions, we obtain a sound criterion for security.

Corollary 7.8. *If $\hat{F}$ as defined in Definition 7.12 has a safe solution, then the program is secure.*

Proof. By Lemma 7.7 any safe solution of $\hat{F}$ restricted to X is a solution of F as defined in Definition 7.11 and by definition it is also a safe solution of F. With Lemma 7.6 and Lemma 7.4 the claim follows directly. □

We conclude this section with Example 7.7 where we illustrate how the above development is capable to verify the security of the program from Example 4.2.

Example 7.7. Consider the program from Example 4.2. In order to verify its security we utilise the location abstraction from Example 7.1, the predicate abstraction from Example 7.3 as state abstraction, the semantic abstraction from Example 7.4, the read write abstraction from Example 7.5 and the abstraction of control dependence from Example 7.6.

In order to search for a safe solution of $\hat{F}$ let $\hat{d}_0 \in \hat{D}$ be such that $\hat{d}_0$ maps $\mathfrak{st}$ to the top element and everything else to bottom. Starting from $\hat{d}_0$ and using the iteration $\hat{d}_{n+1} = \hat{d}_n \sqcup \hat{F}(\hat{d}_n)$ we obtain a pre-fixed point $\hat{d}$ of $\hat{F}$ that lies above $\hat{d}_0$, as the sequence is increasing and $\hat{D}$ finite. In the same manner as in Lemma 7.5 we see that $\hat{d}$ is a solution for $\hat{F}$, as $\hat{d}(\mathfrak{st}) = \top$. What remains to be checked is whether this solution is also safe.

In order to illustrate the propagation by $\hat{F}$ and how it finally manages to assert the security of our program we depict in Figure 7.2 the relevant part of $\hat{d}$ in a graphical form together with an indication how the propagation through $\hat{F}$ takes place. The nodes of the graph represent the extended abstract locations, that is either control locations or pairs consisting of a control location and a variable or a second control location. They are labelled with the abstract state assigned to the extended abstract location by $\hat{d}$ in the first line as well as the abstract location itself in the second line, where we again use only part of the head of a command to denote the corresponding control location. We omit the part corresponding to pairs of two control locations which are used for control dependencies, as there is no propagation happening over those in this example because no branching location in the main copy is reached by a non-trivial state and our abstraction is strict. We also leave out the copy for the variable i as no location where i is written is reached by a non-bottom states in the main copy, such that no non-trivial states are propagated there. The nodes in the respective copies are arranged in the same way as in the graph notation in Example 4.2. Edges denote the propagation by $\hat{F}$ between the nodes. Edges between different copies of the same control location correspond to the propagation of the abstract state itself, while other edges correspond to propagation of the transformed abstract state according to the semantic abstraction f. The emphasised part corresponds to propagation by $\hat{F}$ to parts that reach the main copy again. Parts that cannot propagate back to the main copy or stay bottom throughout the propagation have been greyed out as they play no role in asserting that the solution is safe.

The propagation starts at the initial location $\mathfrak{st}$ in the main copy, which is depicted in the top right. The node is mapped to the top element $\top$ by $\hat{d}$, as this is the initialisation in $\hat{d}_0$ and the iteration is increasing. As the initial location writes all variables in $H = \{h\}$, this state is propagated to **if** b in the copy for h, depicted on the left.

Note that $\hat{F}$ also propagates the top value to $(\mathfrak{st}, \mathfrak{st})$, the copy for control dependencies starting at $\mathfrak{st}$, but as **if** b is the only successor of $\mathfrak{st}$ and post dominates $\mathfrak{st}$, that value is not propagated any further. The same is true for all other locations in the main copy that are reached by non-trivial values, which is why we will ignore them from now on. From the **if** b node in the copy for h the top value is propagated through both branches but only the left branch contains a node that reads h and where propagation back into the main copy is possible. In the left branch i is assigned zero and therefore at the node for `x = h` the propagated state is $\mathit{even}(i)$, which is then propagated to the corresponding node in the main copy. As x is written at that location, the abstract state is propagated to the successor location, the loop head in the copy for x. There the abstract state is propagated into the loop body and strengthened with $i < 2u$, then transformed to $\mathit{odd}(i) \wedge i < 2u$ by the `i=i+1` node, and then propagated back into the main copy from the `z = x` node, as x is read. Note that in the copy for x the local propagation does not fully go through the loop as x is overwritten at the `x = y` node, such that the abstract state $\mathit{even}(i)$ at the loop head is not weakened. From the main copy the abstract state $\mathit{odd}(i) \wedge i < 2u$ is propagated from the `z = x` node into the copy for z where it is propagated unchanged to the `y = z` node and then back into the main copy. The still unchanged state is then propagated into the copy for y, which is the only copy, besides those for control dependencies, from where it could possibly be propagated to the observable `print y` node in the main copy. As the abstract state entails the loop guard $i < 2u$, it is only propagated into the left branch where it is changed by `i=i+1` to $\mathit{even}(i)$ and propagated to the `x = y` node. From there the $\mathit{even}(i)$ state is then propagated back through the main copy to the copy for x where the propagation becomes stable as the $\mathit{even}(i)$ state was already propagated to the head of the while loop in that copy.

Finally, we can observe that the node for `print y` in the main copy, which is the only location in dom(obs), is still mapped to bottom, whereby $\hat{d}$ must be a safe solution by the same argument as in Lemma 7.5 and therefore the program is secure according to Corollary 7.8.

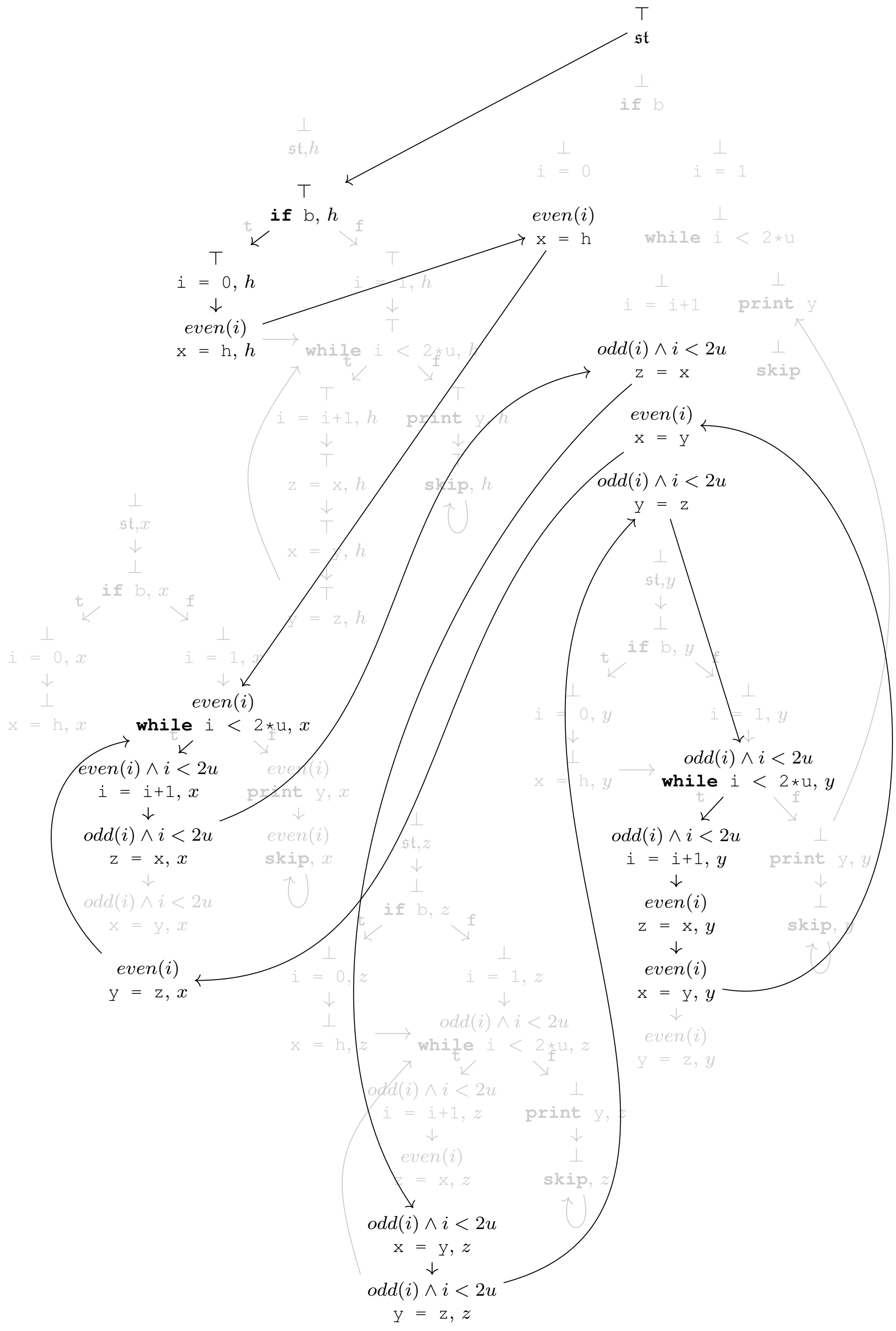


Figure 7.2: Solution of $\hat{F}$ for Example 7.7 with Propagation.

7.3 Regular Abstraction

We now lay out a different approach to exploit our semantic characterisation to obtain sound analyses for our security property. While the approach in Section 7.2 was targeting fixed point–based safety analyses, for which we assumed the analysis to propagate abstract states via abstract transformers that we could freely combine, we now target arbitrary safety analyses that are capable to verify the unreachability of certain error locations in programs. To this end we first construct in Section 7.3.1 for a given program a regular automaton which provides an over-approximation of execution paths corresponding to critical executions. For program analyses that can verify the unfeasibility of a regular set of executions this can directly be exploited to verify the security of a program. We do so in Section 7.3.4 where we describe a prototypical implementation. In Section 7.3.3 we target further analyses by folding the automaton from Section 7.3.1 into the control locations of the program to be analysed in order to obtain a new program together with a set of error locations, whose unreachability in the new program implies the security of the original program. While both approaches target a broader class of safety analyses than the approach in the previous section, they rely on a less precise abstraction of data control dependencies, where we cannot directly exploit the intermediate results of the analysis itself to compute those on the fly during analysis.

7.3.1 Information Flow Automaton

For this section we assume that the set of control locations N of the fixed admissible program (Σ, N, E, $\mathfrak{st}$, $\mathfrak{te}$, $[\![.]\!]$, def, use, L, obs) is finite. We also assume that we are given an *abstraction of controlled writes*, that is a mapping $\mathrm{def}_{\mathrm{cd}} \colon N \to 2^{\mathrm{Var}}$ which fulfils for all i, j, k, σ, v:

$$i \xrightarrow{cd}_\sigma j \xrightarrow{dd_v}_\sigma k \Rightarrow v \in \mathrm{def}_{\mathrm{cd}}(n_i). \tag{7.24}$$

As we assume N to be finite, we can compute an abstraction of controlled writes by traversing E over the non post dominating control locations and collecting all written variables, that is

$$\mathrm{def}_{\mathrm{cd}}(n) = \{v \mid v \in \mathrm{def}(m) \wedge (n,m) \in (E \setminus \{(n',n'') \mid n'' \xrightarrow{pd} n\})^+\}.$$

Definition 7.13. The *information flow automaton* for the fixed program is defined as the finite automaton $\mathcal{A} = (Q, N, \delta, q_0, \{q_f\})$ where the set of states $Q = N \cup \mathrm{Var} \cup \{q_0, q_f\}$ consists of all control locations and variables of the program plus a distinct initial state q_0 and the single accepting state q_f, the alphabet N is the set of program locations and the transition relation $\delta \subseteq Q \times N \times Q$ consists of the following transitions for all $m, n \in N$ and $v \in \mathrm{Var}$:

$$q_0 \xrightarrow{\mathfrak{st}}_\delta \mathfrak{st} \tag{7.25}$$

$$q_0 \xrightarrow{\mathfrak{st}}_\delta v \qquad \text{if } v \in \mathrm{def}(\mathfrak{st}) \cup \mathrm{def}_{\mathrm{cd}}(\mathfrak{st}) \tag{7.26}$$

$$q_0 \xrightarrow{\mathfrak{st}}_\delta q_f \qquad \text{if } \mathfrak{st} \in \mathrm{dom}(\mathrm{obs}) \tag{7.27}$$

$$m \xrightarrow{n}_\delta m \qquad \text{if } \neg\, n \xrightarrow{pd} m \tag{7.28}$$

$$m \xrightarrow{n}_\delta v \qquad \text{if } \neg\, n \xrightarrow{pd} m \wedge v \in \mathrm{def}(n) \cup \mathrm{def}_{\mathrm{cd}}(n) \tag{7.29}$$

$$m \xrightarrow{n}_\delta q_f \qquad \text{if } \neg\, n \xrightarrow{pd} m \wedge n \in \mathrm{dom}(\mathrm{obs}) \tag{7.30}$$

$$v \xrightarrow{n}_\delta v \qquad \text{if } v \notin \mathrm{def}(n) \tag{7.31}$$

$$v \xrightarrow{n}_\delta n \qquad \text{if } v \in \mathrm{use}(n) \tag{7.32}$$

$$v \xrightarrow{n}_\delta w \qquad \text{if } v \in \mathrm{use}(n) \wedge w \in \mathrm{def}(n) \cup \mathrm{def}_{\mathrm{cd}}(n) \tag{7.33}$$

$$v \xrightarrow{n}_\delta q_f \qquad \text{if } v \in \mathrm{use}(n) \wedge n \in \mathrm{dom}(\mathrm{obs}) \tag{7.34}$$

The information flow automaton does not keep track of the control flow itself and is intended to be combined with a suitable abstraction of control flow, as we will do when deriving the security program in Section 7.3.3. In order to motivate the definition we describe the conditions under which prefixes of control location sequences from executions in R and C are accepted by the individual states of the automaton. As R is defined inductively along data, control and data control dependencies the automaton checks that the conditions corresponding to the currently tracked dependency hold while accepting a prefix. By which state of the automaton a prefix is accepted therefore depends upon the last dependency that is still open or has just been completed in the prefix. If the prefix ends with a complete data or data control dependency, it shall be accepted by the state corresponding to the reached location. If the prefix ends in a non complete data or data control dependency, the prefix shall be accepted by the state corresponding to the variable used by the dependency. If the prefix ends in a control dependency there is no need to distinguish between open and closed control dependencies, due to the prefix property of control dependencies (Lemma 6.10) and by also exploiting the transitivity of control dependencies (Lemma 6.11) the automaton shall accept the prefix with the location corresponding to the first index upon which the reached index is control dependent. Finally, if the prefix of the execution corresponds to a critical state index pair from C, it shall be accepted by the final state q_f. We formalise this in the proof of the following theorem, which states the soundness of the information flow automaton for our purposes.

Theorem 7.9. $(\sigma, i) \in C \Rightarrow (n_k)_{k \leq i} \in \mathcal{L}(\mathcal{A})$

Proof. For the proof we first establish the following properties for all $\sigma, i, j, v, \sigma', i'$:

$$i \xrightarrow{cd}_\sigma j \Rightarrow n_i \xrightarrow{(n_k)_{i<k<j}}_\delta n_i \tag{7.35}$$

$$i \xrightarrow{dd_v}_\sigma j \Rightarrow v \xrightarrow{(n_k)_{i<k<j}}_\delta v \tag{7.36}$$

$$i \xrightarrow{dcd_v^{\sigma',i'}}_\sigma j \Rightarrow v \xrightarrow{(n_k)_{i<k<j}}_\delta v \tag{7.37}$$

We observe that in the case where $j = i + 1$ the right hand-sides degenerate to $q \xrightarrow{\epsilon}_\delta q$, which holds trivially. Otherwise we have (7.35), as from the definition of control dependence we have $\forall k \in (i, j]\colon \neg\, n_k \xrightarrow{pd} n_i$ and thereby have $\forall k \in (i, j)\colon n_i \xrightarrow{n_k}_\delta n_i$ by (7.28). For (7.36) and (7.37) we note that in either case we have $\forall k \in (i, j)\colon v \notin \text{def}(n_k)$, either by the definition of data dependence in the first or data control dependence in the second case and therefore have $\forall k \in (i, j)\colon v \xrightarrow{n_k}_\delta v$ by (7.31).

We now prove by induction over the definition of R (Definition 5.7) that we have the following property for all σ and i:

$$(\sigma, i) \in R \Rightarrow (n_k)_{k \leq i} \in \mathcal{L}_\mathcal{A}(n_i) \vee \exists \iota\colon \iota \xrightarrow{cd}_\sigma i \wedge (n_k)_{k \leq i} \in \mathcal{L}_\mathcal{A}(n_\iota) \tag{7.38}$$

In the base case (5.6) we have $i = 0$ and $(n_k)_{k \leq i} = (\mathfrak{st})$. By (7.25) we have $q_0 \xrightarrow{\mathfrak{st}}_\delta \mathfrak{st}$ and thereby $(\mathfrak{st}) \in \mathcal{L}_\mathcal{A}(\mathfrak{st})$ as required.

In the case (5.7), where $(\sigma, j) \in R$ due to $i \xrightarrow{dd_v}_\sigma j$ with $(\sigma, i) \in R$, we have from the induction hypothesis that either $(n_k)_{k \leq i} \in \mathcal{L}_\mathcal{A}(n_i)$ or we obtain the existence of an index ι such that $\iota \xrightarrow{cd}_\sigma i$ and $(n_k)_{k \leq i} \in \mathcal{L}_\mathcal{A}(n_\iota)$. Due to $i \xrightarrow{dd_v}_\sigma j$ we have $v \in \text{def}(n_i)$. We show that in both cases we have $(n_k)_{k \leq i} \in \mathcal{L}_\mathcal{A}(v)$, as then we obtain with (7.36) that $(n_k)_{k<j} \in \mathcal{L}_\mathcal{A}(v)$ and thereby, as $v \in \text{use}(n_j)$, further obtain with (7.32) that $(n_k)_{k \leq j} \in \mathcal{L}_\mathcal{A}(n_j)$ as required.

First assume that it is the case that $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(n_i)$. We then obtain the existence of a state q such that $(*)$: $q \xrightarrow{n_i}_{\delta} n_i$ and $(n_k)_{k<i} \in \mathcal{L}_{\mathcal{A}}(q)$, which, as $i > 0$, implies $q \neq q_0$. There then remain two options for q, either q is a control location and $(*)$ holds due to transition (7.28), in which case we can instead take transition (7.29), as $v \in \mathrm{def}(n_i)$, to obtain $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(v)$, or q is a variable and $(*)$ holds due to transition (7.32), in which case we can again with $v \in \mathrm{def}(n_i)$ use (7.33) instead to also obtain $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(v)$ and close the fist case where $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(n_i)$.

In the second case where $\iota \xrightarrow{cd}_{\sigma} i$ and $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(n_\iota)$, we have $(n_k)_{k\leq i} = (n_k)_{k<i} \cdot n_i$ and obtain q such that $(n_k)_{k<i} \in \mathcal{L}_{\mathcal{A}}(q)$ and $q \xrightarrow{n_i}_{\delta} n_\iota$. We might assume without loss of generality that $n_\iota \neq n_i$ as otherwise this would bring us back to the first case. Thereby the only fitting rule is (7.28) and it must be the case that $q = n_\iota$ and $(n_k)_{k<i} \in \mathcal{L}_{\mathcal{A}}(n_\iota)$. We can then apply rule (7.29) instead to also obtain $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(v)$ and close the second case, which closes the case for (5.7).

The case (5.9), where $(\sigma, j) \in R$ due to $i \xrightarrow{dcd_v^{\sigma',i'}}_{\sigma} j$ with $(\sigma, i) \in R$ and $(\sigma', i') \in R$, works completely analogously by utilising $v \in \mathrm{def}_{\mathrm{cd}}(n_i)$ and (7.37) instead.

Finally, in the case of (5.8), where $(\sigma, j) \in R$ due to $i \xrightarrow{cd}_{\sigma} j$ with $(\sigma, i) \in R$, we have again by induction hypothesis that either $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(n_i)$ or we obtain the existence of ι such that $\iota \xrightarrow{cd}_{\sigma} i$ and $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(n_\iota)$. In the first case we define $\iota = i$ and in the second case utilise the transitivity of control dependencies to obtain in either case that $\iota \xrightarrow{cd}_{\sigma} j$ and $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(n_\iota)$. With (7.35) we then directly obtain $(n_k)_{k<j} \in \mathcal{L}_{\mathcal{A}}(n_\iota)$. As $\neg\, n_j \xrightarrow{pd} n_\iota$ because of $\iota \xrightarrow{cd}_{\sigma} j$, we obtain $(n_k)_{k\leq j} \in \mathcal{L}_{\mathcal{A}}(n_\iota)$ by rule (7.28) as required.

With this we can now conclude the proof and to that end assume that $(\sigma, i) \in C$. As $(\sigma, i) \in R$ we have from (7.38), that $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(n)$ for some $n \in N$. As we did in the case for (5.7) we consider the last transition that was applied in order to obtain $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(n)$. Due to $n \in N$ this last transition must be one of (7.25), (7.28) or (7.32). In each case we can use the fact that $n_i \in \mathrm{dom}(\mathrm{obs})$ and instead apply (7.27), (7.30) or (7.34) respectively to obtain $(n_k)_{k\leq i} \in \mathcal{L}_{\mathcal{A}}(q_f) = \mathcal{L}(\mathcal{A})$ as required. □

With this result we can use any analysis that is capable of verifying that no sequence of locations from $\mathcal{L}(\mathcal{A})$ is the prefix of an actual execution to assert the security of a program as recorded by the following corollary.

Corollary 7.10. *If* $\mathcal{L}(\mathcal{A}) \cap \big\{(n_k)_{k\leq i} \mid \sigma \in \Sigma, i \in \mathbb{N}\big\} = \emptyset$ *then the program is secure.*

Proof. This follows directly from Theorem 7.9 in combination with Corollary 5.5. □

7.3.2 Precision Compared to the PDG

One will usually want to utilise an analysis that checks an over-approximation of the set of feasible paths for acceptance by the information flow automaton to verify the security of a program with this approach. Unlike the upper bound established by the soundness properties proven before, we now give a lower bound for the precision of this approach for analyses that are at least as precise as the control flow abstraction itself. The following lemma to this end states, that if the used abstraction of controlled writes is at least as precise as the control flow abstraction, which for instance is the case for the one we defined at the beginning of the previous section, then any execution path in the control flow abstraction that is accepted by the information flow automaton reaches an observable control location in the forward slice of the PDG. Therefore, for such abstractions this approach is at least as precise as the PDG-based approach from Section 7.1.

Lemma 7.11. *If the abstraction of controlled writes is precise with respect to the control flow abstraction, that is for all $v \in \mathrm{Var}$ and $n \in N$ it holds that*

$$v \in \mathrm{def}_{\mathrm{cd}}(n) \Rightarrow \exists \pi, i \colon n = n_0 \wedge 0 \xrightarrow{cd}_{\pi} i \wedge v \in \mathrm{def}(n_i), \tag{7.39}$$

then the information flow automaton is as least as precise as the PDG, that is for all initial paths π in the control flow abstraction and all indices k it holds that

$$\pi_k \in \mathcal{L}(\mathcal{A}) \Rightarrow n_k \in R_{PDG} \cap \mathrm{dom}(\mathrm{obs}). \tag{7.40}$$

Proof. In order to prove the lemma we strengthen (7.40) by adding cases for the non accepting states of the information flow automaton. We prove the following three implications simultaneously via induction over the index k:

$$\pi_k \in \mathcal{L}_{\mathcal{A}}(n) \Rightarrow n \in R_{PDG} \wedge (n = n_k \vee \exists i < k \colon n_i = n \wedge i \xrightarrow{cd}_{\pi} k) \tag{7.41}$$

$$\pi_k \in \mathcal{L}_{\mathcal{A}}(v) \Rightarrow$$

$$\exists i \leq k \colon n_i \in R_{PDG} \wedge v \in \mathrm{def}(n_i) \cup \mathrm{def}_{\mathrm{cd}}(n_i) \wedge \forall j \in (i, k] \colon v \notin \mathrm{def}(n_j) \tag{7.42}$$

$$\pi_k \in \mathcal{L}_{\mathcal{A}}(q_f) \Rightarrow n_k \in R_{PDG} \cap \mathrm{dom}(\mathrm{obs}) \tag{7.43}$$

Let the implications hold for all indices smaller than k and let $\pi_k \in \mathcal{L}_{\mathcal{A}}(q')$ hold for some $q' \in Q$. We perform a case distinction over the last transition $q \xrightarrow{n_k}_\delta q'$ with which $\pi_k \in \mathcal{L}_\mathcal{A}(q')$ can be deduced, that is based on rules (7.25)–(7.34) of Definition 7.13.

In the cases where the transition corresponds to (7.25)–(7.27), that is if $q = q_0$, we have $k = 0$ and $n_k = \mathfrak{st}$, whereby $n_k \in R_{PDG}$ holds directly. The additional conditions required by (7.41)–(7.43) respectively follow from the corresponding side conditions of (7.25)–(7.27).

In the cases where $q = m \in N$, that is rules (7.28)–(7.30), we obtain from the induction hypothesis for $k-1$ by (7.41) that $m \in R_{PDG}$ and either $n_{k-1} = m$ or we obtain an index i such that $n_i = m$ and $i \xrightarrow{cd}_\pi k-1$. In the former case we let $i = k-1$ and due to the side condition that $\neg\, n_k \xrightarrow{pd} m$, which is present for all rules in this case, we obtain in either of these cases that $i \xrightarrow{cd}_\pi k$ and thereby also that $n_k \in R_{PDG}$ from $n_i = m \in R_{PDG}$.

If we consider the case of rule (7.28), where $q' = m$ and $m \xrightarrow{n_k}_\delta m$, we have (7.41) due to $m \in R_{PDG}$ and $i \xrightarrow{cd}_\pi k$ with $n_i = m$. The other implications are trivial as the antecedent is false.

The case of rule (7.29), where $q' = v \in \mathrm{Var}$, follows as only implication (7.42) is non-trivial and we can instantiate the quantified i with k and have $n_k \in R_{PDG}$ and $v \in \mathrm{def}(n_k) \cup \mathrm{def}_{\mathrm{cd}}(n_k)$ from the side condition of (7.29).

The case of rule (7.30), where $q' = q_f$, follows as only implication (7.43) is non-trivial and we have $n_k \in R_{PDG}$ as well as $n_k \in \mathrm{dom}(\mathrm{obs})$ from the side condition of (7.30).

In the cases where $q = v \in \mathrm{Var}$, that is rules (7.31)–(7.34), we obtain by the induction hypothesis for $k-1$ from (7.42) the existence of an index $i < k$ such that $n_i \in R_{PDG}$, $v \in \mathrm{def}(n_i) \cup \mathrm{def}_{\mathrm{cd}}(n_i)$ and $\forall j \in (i,k)\colon v \notin \mathrm{def}(n_j)$. The case for rule (7.31) where $v \xrightarrow{n_k}_\delta v$ with side condition $v \notin \mathrm{def}(n_k)$ is trivial, as we have (7.42) using the index i obtained above and the side condition allows us to expand the universal quantifier at the end to k. For the cases (7.32)–(7.34) we first assume the common side condition $v \in \mathrm{use}(n_k)$ and will prove that $n_k \in R_{PDG}$. With this in each case the respective additional side conditions are sufficient to directly obtain (7.41)–(7.43).

In order to show $n_k \in R_{PDG}$ we distinguish the cases where $v \in \mathrm{def}(n_i)$ and those where $v \in \mathrm{def}_{\mathrm{cd}}(n_i)$. If $v \in \mathrm{def}(n_i)$ we have with $v \in \mathrm{use}(n_k)$ that $i \xrightarrow{dd_v}_\pi k$ and thereby with $n_i \in R_{PDG}$ by Definition 7.2 that $n_k \in R_{PDG}$ as required. If $v \in \mathrm{def}_{\mathrm{cd}}(n_i)$ we obtain π' and j' from (7.39) such that $n'_0 = n_i$, $0 \xrightarrow{cd}_{\pi'} j'$ and $v \in \mathrm{def}(n'_{j'})$. We might assume that π' reaches $\mathfrak{te}$ after j'. If $i \xrightarrow{cd}_\pi k$ we directly obtain $n_k \in R_{PDG}$ from $n_i \in R_{PDG}$ otherwise, if it is not the case that $i \xrightarrow{cd}_\pi k$ it cannot be the case that

$n_i = \mathfrak{te}$ due to Lemma 6.7 and Assumption 4 of Definition 4.2. Therefore as π' reaches $\mathfrak{te}$ and $\mathfrak{te} \xrightarrow{pd} n_i = n'_0$ there exists a smallest index l' such that $n'_{l'} \xrightarrow{pd} n'_0$. As $0 \xrightarrow{cd}_{\pi'} j'$, it must be the case that $j' < l'$. As l' is the first index that visits a post dominator of n'_0, all smaller indices are control dependent upon 0 and we might therefore assume that j' is the greatest index below l' such that $v \in \mathrm{def}(n'_{j'})$. As it is not the case that $i \xrightarrow{cd}_{\pi} k$ we can also fix the smallest index l greater than i such that $n_l \xrightarrow{pd} n_i$, which must be smaller or equal to k. It must be the case that $n_l = n'_{l'}$, as otherwise one of them would not post dominate the other and we could construct a path from n_i over the latter to $\mathfrak{te}$ without visiting the former. We therefore obtain a new path $\hat{\pi}$ by following π' until l' and then switching to π from $l + 1$ on, that is we let $\hat{\pi} = \pi'_{l'} \cdot (\pi \ll l + 1)$. We have that $\hat{n}_0 = n_i$ and $\hat{n}_{k-l+l'} = n_k$. As $j' < l'$, we have $0 \xrightarrow{cd}_{\hat{\pi}} j'$ such that with $n_i \in R_{PDG}$ we obtain $\hat{n}_{j'} \in R_{PDG}$. Because j' was the last definition of v before l' in π' and we also obtained from the induction hypothesis for $k - 1$ that there is no definition of v in π between i and k, we have $j' \xrightarrow{dd_v}_{\hat{\pi}} k - l + l'$ and thereby $n_k = \hat{n}_{k-l+l'} \in R_{PDG}$ as required. □

7.3.3 Security Program

While directly verifying that no execution of the original program corresponds to a path accepted by the information flow automaton is the approach used by our prototypical implementation, which we describe in Section 7.3.4, here we first describe another approach where we fold the automaton into the program itself to obtain a new program together with a simple safety property that might be targeted by a broad class of safety analyses. In order to fold the automaton into the program one option is to utilise pairs of program locations and automata states as locations in the derived program, which yields a non-deterministic program. We pursue another approach where we directly perform a power set construction and use sets of automaton states instead of simple states in order to preserve determinism. In this setting we choose this option as it actually yields smaller programs in our examples, as the control flow greatly limits the number of reachable states. The construction is however exponential in theory and while not the case in our examples, it is not hard to construct examples where the exponential blow-up can be observed. In practice one might therefore instead opt for the non-deterministic option based on the program and analysis at hand. We call the program constructed in this way the corresponding security program.

Definition 7.14. For a given program $(\Sigma, N, E, \mathfrak{st}, \mathfrak{te}, [\![.]\!], \mathrm{def}, \mathrm{use}, L, \mathrm{obs})$ and corresponding information flow automaton $\mathcal{A} = (Q, N, \delta, q_0, \{q_f\})$ the corresponding *security program* is defined as $(\Sigma, \hat{N}, \hat{E}, \hat{\mathfrak{st}}, \hat{N}_f, (\!|.|\!))$ where

$$\begin{aligned}
\hat{N} =& N \times 2^Q \text{ is the set of control locations,}\\
\hat{E} =& \big\{ ((n, \hat{Q}), (n', \delta(\hat{Q}, n'))) \mid (n, n') \in E \big\} \text{ is the control flow abstraction,}\\
\hat{\mathfrak{st}} =& \big(\mathfrak{st}, \delta(q_0, \mathfrak{st})\big) \text{ is the initial control location,}\\
\hat{N}_f =& \big\{ (n, \hat{Q}) \in \hat{N} \mid q_f \in \hat{Q} \big\} \text{ is the set of critical control locations, and}\\
(\!|(n, \hat{Q}), \sigma|\!) =& \big(([\![n, \sigma]\!]_l, \delta(\hat{Q}, [\![n, \sigma]\!]_l)), [\![n, \sigma]\!]_s\big) \text{ is the semantic step function, using}\\
& \delta(\hat{Q}, n) = \{\, q' \mid q \in \hat{Q} \wedge q \xrightarrow{n} q' \,\}.
\end{aligned}$$

We define executions of the security program in the same manner as for our main program model from Definition 4.2, where for any input state the semantic step function is repeatedly applied starting at the initial control location, that is $((\!|\hat{\mathfrak{st}}, \sigma|\!)^i)_{i \geq 0}$ is the execution corresponding to the initial state $\sigma \in \Sigma$. The safety property to be targeted by analyses is then whether any such execution can at any point reach a control location from $\hat{N}_f$. If no such execution exists the original program is secure as stated in the following theorem.

Theorem 7.12. *If no execution of the security program as defined by Definition 7.14 reaches a location from $\hat{N}_f$ then the original program is secure.*

Proof. Assume the program is not secure. According to Corollary 5.5 there then exists a critical observable execution, that is $(\sigma, i) \in C$. From this we then have by Theorem 7.9 that $(n_k)_{k \leq i} \in \mathcal{L}(\mathcal{A})$ and obtain a sequence $(q_k)_{0 < k \leq i}$ such that $q_i = q_f$ and $\forall k < i \colon q_k \xrightarrow{n_k} q_{k+1}$. The initial state σ gives rise to an execution of the security program, which has the form $((n_k, \hat{Q}_k), \sigma_k)_{0 \leq k}$ and for which we observe inductively that $\forall k \leq i \colon q_k \in \hat{Q}_k$. As $q_i = q_f$ we have that the execution reaches a location from $\hat{N}_f$. □

As the initial paths in the control flow abstraction of the security program that reach N_f correspond to the initial path in the control flow abstraction of the original program that are accepted by the information flow automaton, we directly obtain the following corollary about the precision of the security program from the result about the precision of the information flow automaton, Lemma 7.11.

Corollary 7.13. *If the program is secure according to the PDG as defined in Section 7.1 and the abstraction of controlled writes underlying the information flow automaton is precise with respect to the control flow, then no state from $\hat{N}_f$ is reachable from $\hat{\mathfrak{st}}$ in $(\hat{N}, \hat{E})$.*

We conclude this section with a concrete example.

Example 7.8. Consider the program for the command pictured in Figure 7.3 with $H = \{h\}$. The program is a condensed example of how a program might utilises data properties (in this case $p = 0$) to guide control flow to avoid observable behaviour when handling confidential data. From the semantics it is straightforward to see that the attacker can only observe the initial value of y if c is initially zero and p is not and can make no observation otherwise, wherefore the program is secure.

```
if c then x = h; p = 0 else x = y fi;
if p then print x fi
```

Figure 7.3: Secure program for $H = \{h\}$.

For the approach from this section to verify the security of this program we require an abstraction of controlled writes that we obtain as defined in the previous section by collecting the variables written within each control structure, which in this example are only x and p in the first if-then-else command. With this we let $\text{def}_{\text{cd}}($**if** c$) = \{x, p\}$ and $\text{def}_{\text{cd}}(n) = \emptyset$ for all other n. We can then obtain the corresponding security program via Definition 7.14. Based on the unoptimised definition the program contains several structurally unreachable locations which are irrelevant for reachability of the critical locations $\hat{N}_f$. The structurally reachable part of the security program is depicted in Figure 7.4.

In order to verify the security of the original program, it suffices, according to Theorem 7.12, to show that the annotated **print** x node in the right branch is not reachable. To do so is rather simple for a safety analysis, it only has to be capable of propagating the information that p is zero from the p = 0 node to the following **if** p node and deducing that the *then*-branch cannot be taken.

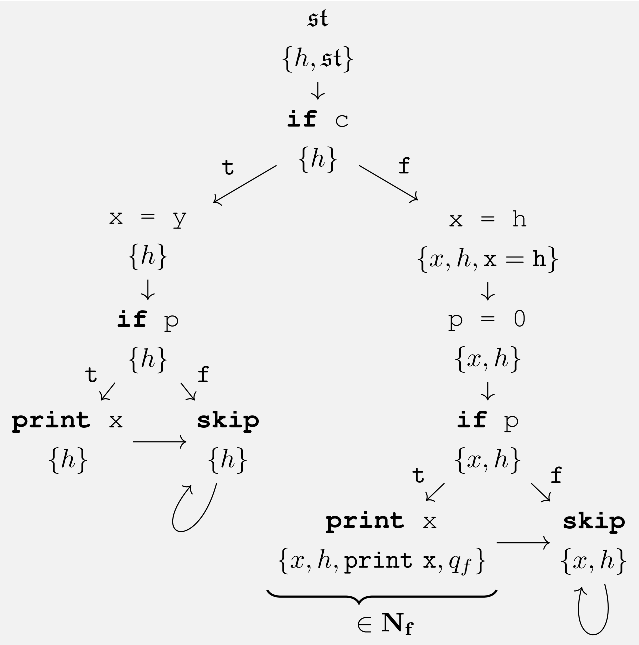


Figure 7.4: Security program for the program from Figure 7.3.

7.3.4 Implementation

For a basic experimental evaluation of our overall approach we developed a prototypical implementation based on the regular abstraction from Definition 7.13 utilising the software verification platform *CPAchecker*[1] by Beyer and Keremoglu [6]. While the platform targets C as well as Java programs, our implementation is only meant as a demonstrator and we restricted ourselves to a small intraprocedural fragment corresponding to our command language from Section 4.3 that does not support threading, heap access, function calls and several other features present in the intermediate representation. As the intermediate representation used by the platform utilises edge annotated control flow graphs instead of the node annotated graphs used in our development, the necessary definitions have to be translated adequately. This can be done rather straightforwardly and we will not go into detail about that here. In a first phase the implementation computes the post dominators and an abstraction of controlled

[1] `https://cpachecker.sosy-lab.org`

writes. Based on this the main analysis is performed in a second phase. The analysis by the platform works by allowing multiple analyses to run in parallel, each propagating and transforming its own internal states over provided transitions of the program in such a way that each analysis can restrict the set of allowed transitions or mark certain states as error states, similar to how a product automaton for the intersection of regular languages works. If an error state is reached by some analysis the platform attempts to perform refinements in the individual analyses to prove it unreachable or finally fails if that is not possible. This allows us to implement an automaton corresponding to the one from Definition 7.13 directly as one individual analysis which keeps track of the state of the automaton, updates them according to the corresponding transitions and reports an error when the accepting state of the automaton is reached. We can then combine our analysis with different safety analyses provided by the platform to evaluate the main goal of our approach, that is how increasingly precise safety analyses can be exploited to obtain increasingly precise security analyses. To this end we test our analysis in two configurations combined with analyses provided by the platform. Firstly as a reference point, we combine our analysis solely with a so called *location analysis* that only keeps track of the control location. With this we obtain an analysis which has the same power as a simple PDG-based analysis. This also allows us to directly compare results obtainable by our methodology with results which would be obtained by classical methods. Secondly we utilise a so called predicate analysis [7], which works similar to the predicate abstractions we utilised for our fixed point–based examples in Section 7.2. In this configuration our implementation can handle examples similar to those given so far like Example 5.7, Example 7.8 or Example 7.9 given below.

Example 7.9. The program in Figure 7.5 is secure. Translated to an equivalent C program, this can be verified by our implementation utilising either the predicate analysis or the interval analysis that is available on the platform as well. When only using a location analysis the verification fails as one would also expect from syntactic approaches like PDGs or type systems.

```
if b then
  j = 100; x = h
else
  j = 1000
fi;
i = 0;
while i < j do
  i = i + 1
  if i > 100 then
    y = x
  fi
od;
print y
```

Figure 7.5: Secure program for $H = \{h\}$.

Experimental Evaluation. For evaluation we utilised 282 program snippets that ship with the platform for unit test and benchmark purposes. As security specifications we generated all possible pairs of local variables used within each snippet and if there were more than one thousand possible pairs we randomly sampled one thousand of them. In this way we obtained a total of 168 491 queries for our analysis where we defined the first variable as high and all writes to the second variable as observable. The analysis of many of those queries either encountered unsupported operations during analysis, hit the timeout of 5 minutes or caused the platform to report an error for other reasons. In a first step we were left with 25 985 successfully handled queries when using nothing but the location analysis of the platform. Of the 25 985 handled queries 24 989 were deemed secure and 996 insecure by this configuration of the analysis, which as mentioned corresponds to a simple PDG-based analysis. In a second step we utilised a predicate analysis with our approach and of the 25 985 queries handled in the first step 25 117 (an increase of 128) were deemed secure while 560 (a decrease of 436) were deemed insecure and an additional 308 queries hit the timeout or produced an error. When only considering the queries that did not produce an error or hit the time out in the first configuration this leaves us with 868 (a decrease of 128) queries for which security could not be asserted by this configuration of the analysis, which is a 13 % improvement over the 996 queries whose security could not be verified initially in this subset.

When interpreting these numbers one should keep in mind that the selection of programs will probably be heavily biased as they were selected as test cases for the platform. Also we do not have any ground truth for our generated queries so we cannot determine how many of the failed queries were due to actual flows. Yet the results indicate that there might be a significant number of cases where our approach can successfully verify security while syntactic approaches fail.

8 Conclusion

We have presented a novel approach for static information flow control that can harness the power of modern safety analyses. The development is based on a simple yet versatile program model that is not limited to finite control or data. The program model can handle procedures for instance by utilising a control stack as control locations and the availability of infinite data as well as branching also allows representing local variables or arrays. The targeted security property is in general termination insensitive while still providing some guarantees for non-terminating executions as it allows for observations during executions but strongly limits how non-terminating executions might leak information about confidential inputs. After observing that the tracking of data and control dependencies within single executions is insufficient for ensuring information flow security we developed our central characterisation of pairs of executions breaking the security property based on the tracking of data and control dependencies within those pairs. The characterisation is inductive and precisely describes matching points in executions to which confidential input is propagated and where different data is read. As the characterisation utilises the full semantics, it allows analyses to exploit arbitrary information about the executions of a program without risking loss of soundness. We utilised the characterisation to derive a single execution approximation which can be targeted by safety analyses. The single execution approximation additionally tracks data control dependencies to handle flows not directly visible in an execution itself when only tracking data and control dependencies. We provided rigorous soundness proofs that have also been formalised and machine checked using the interactive theorem prover Isabelle/HOL. Based on the single execution approximation we then described multiple possible applications of how this approximation might be exploited to obtain effective security analyses that are providing different interfaces for generic safety analyses.

In a first application we described how PDGs can directly be derived as a crude approximation of our characterisation. We revisited a definition of PDGs from the literature adequate for our program model and provided a formal connection to our approach allowing us to derive a corresponding soundness result for the PDG-based approach as well as a reference point for the precision of our development.

As a second application we used abstract interpretation–like techniques to derive a fixed point–based approach that provides an interface to generic abstract domains. This approach allows for a deeper integration with the underlying safety analyses as it can reuse intermediate results from the safety analyses to improve the information flow approximation. In a concrete example we employed a predicate abstraction to verify the security of a program using this approach.

In a third application we derived a regular approximation that describes a regular language of critical execution paths for programs with finite control. The approximation can be directly targeted by existing safety analyses by verifying that there is no feasible execution of the program taking such a critical path. Alternatively the regular approximation can be folded into the control structure of the original program to obtain a derived program together with a simple control location reachability property whose validity implies the security of the original program. We also prove that this approach is at least as precise as the PDG-based approach for our program model. Finally, we reported empirical results from a prototypical implementation based on the regular approximation that indicate practical benefits of our approach.

Future Work. Possible future work includes further empirical evaluation of this approach, in particular comparing its efficiency and precision against approaches based on self composition.

As already mentioned in Chapter 2 it might also be possible to exploit our central characterisation of pairs of critical executions to derive an optimised self-composition where the employed safety analysis has to consider fewer execution pairs and can verify a simpler reachability property while still exploiting the possible advantages of a self composition–based approach.

Also of interest would be the study of possible extensions of the program model or its applicability for further programming features. While procedures and arrays can be handled rather straightforwardly by the existing program model, it is less clear whether suitable representations for features like objects and pointers exist.

Of interest could also be the potential development of a staggered approach utilising this development where increasingly precise analyses might be optimised using information from less precise analyses run before exploiting the common characterisation of critical executions. Such an approach could for instance begin with a PDG-based analysis and then employ more expensive but more precise analyses only to check possibly spurious paths found in the PDG.

Studying whether this approach can also be adapted for the verification of concurrent programs is another possible direction for future research.

Bibliography

[1] Hiralal Agrawal and Joseph Robert Horgan. "Dynamic Program Slicing". In: *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990.* Ed. by Bernard N. Fischer. ACM, 1990, pp. 246–256. DOI: 10.1145/93542.93576.

[2] Rajeev Alur, Pavol Cerný, and Steve Zdancewic. "Preserving Secrecy Under Refinement". In: *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II.* Ed. by Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener. Vol. 4052. Lecture Notes in Computer Science. Springer, 2006, pp. 107–118. DOI: 10.1007/11787006_10.

[3] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. "Secure information flow by self-composition". In: *Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252. DOI: 10.1017/S0960129511000193.

[4] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. "Static Analysis and Verification of Aerospace Software by Abstract Interpretation". In: *Found. Trends Program. Lang.* 2.2-3 (2015), pp. 71–190. DOI: 10.1561/2500000002.

[5] Dirk Beyer, Matthias Dangl, and Philipp Wendler. "A Unifying View on SMT-Based Software Verification". In: *Journal of Automated Reasoning* 60.3 (2018), pp. 299–335. DOI: 10.1007/s10817-017-9432-6.

[6] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings.* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16.

[7] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. “Predicate abstraction with adjustable-block encoding”. In: *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. Ed. by Roderick Bloem and Natasha Sharygina. IEEE, 2010, pp. 189–197. URL: `http://ieeexplore.ieee.org/document/5770949/`.

[8] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. “Reactive Noninterference”. In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS ’09. Chicago, Illinois, USA: ACM, 2009, pp. 79–90. DOI: `10.1145/1653662.1653673`.

[9] Olivier Bouissou, Eric Conquet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Khalil Ghorbal, Eric Goubault, David Lesens, Laurent Mauborgne, Antoine Miné, Sylvie Putot, Xavier Rival, and Michel Turin. “Space Software Validation using Abstract Interpretation”. In: *Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009)*. Vol. SP-669. Istambul, Turkey: ESA, May 2009, pp. 1–7.

[10] Patrick Cousot and Radhia Cousot. “Static determination of dynamic properties of programs”. In: *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, pp. 106–130. URL: `https://www.di.ens.fr/~cousot/COUSOTpapers/ISOP76.shtml`.

[11] Patrick Cousot and Radhia Cousot. “Systematic Design of Program Analysis Frameworks”. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’79. San Antonio, Texas: ACM, 1979, pp. 269–282. DOI: `10.1145/567752.567778`.

[12] Patrick Cousot and Nicolas Halbwachs. “Automatic Discovery of Linear Restraints Among Variables of a Program”. In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski. ACM Press, 1978, pp. 84–96. DOI: `10.1145/512760.512770`.

[13] David Delmas and Jean Souyris. “Astrée: From Research to Industry”. In: *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*. Ed. by Hanne Riis Nielson and Gilberto Filé.

Vol. 4634. Lecture Notes in Computer Science. Springer, 2007, pp. 437–451. DOI: 10.1007/978-3-540-74061-2_27.

[14] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. “Automating Regression Verification”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE ’14. ACM, 2014, pp. 349–360. DOI: 10.1145/2642937.2642987.

[15] Jérôme Feret. “Static Analysis of Digital Filters”. In: *European Symposium on Programming (ESOP’04)*. LNCS 2986. Springer, 2004. DOI: 10.1007/978-3-540-24725-8_4.

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The program dependence graph and its use in optimization”. In: *International Symposium on Programming*. Ed. by M. Paul and B. Robinet. Berlin, Heidelberg: Springer, 1984, pp. 125–132. DOI: 10.1007/3-540-12925-1_33.

[17] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Transactions on Programming Languages and Systems* 9.3 (July 1987), pp. 319–349. DOI: 10.1145/24039.24041.

[18] Andrea Flexeder, Markus Müller-Olm, Michael Petter, and Helmut Seidl. “Fast Interprocedural Linear Two-variable Equalities”. In: *ACM Transactions on Programming Languages and Systems* 33.6 (Jan. 2012), 21:1–21:33. DOI: 10.1145/2049706.2049710.

[19] Dennis Giffhorn. “Slicing of Concurrent Programs and its Application to Information Flow Control”. PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, 2012.

[20] Dennis Giffhorn and Gregor Snelting. “A new algorithm for low-deterministic security”. In: *Int. Journal of Information Security* 14.3 (2015), pp. 263–287. DOI: 10.1007/s10207-014-0257-6.

[21] Joseph A. Goguen and José Meseguer. “Security Policies and Security Models”. In: *IEEE Symposium on Security and Privacy*. Los Alamitos, CA, USA: IEEE Computer Society, 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014.

[22] Philippe Granger. “Static Analysis of Linear Congruence Equalities Among Variables of a Program”. In: *Proceedings of the International Joint Conference on Theory and Practice of Software Development on Colloquium on Trees in Algebra and Programming (CAAP ’91): Vol 1.* TAPSOFT ’91. Brighton, United Kingdom: Springer, 1991, pp. 169–192. DOI: `10.1007/3-540-53982-4_10`.

[23] Christian Hammer and Gregor Snelting. “Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs”. In: *Int. Journal of Information Security* 8.6 (2009), pp. 399–422. DOI: `10.1007/s10207-009-0086-1`.

[24] Susan Horwitz, Thomas Reps, and David Binkley. “Interprocedural Slicing Using Dependence Graphs”. In: *ACM Transactions on Programming Languages and Systems* 12.1 (Jan. 1990), pp. 26–60. DOI: `10.1145/77606.77608`.

[25] C. Samuel Hsieh, Elizabeth A. Unger, and Ramón A. Mata-Toledo. “Using program dependence graphs for information flow control”. In: *Journal of Systems and Software* 17.3 (1992), pp. 227–232. DOI: `10.1016/0164-1212(92)90111-V`.

[26] Jan Jürjens. “Secrecy-Preserving Refinement”. In: *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings.* Ed. by José Nuno Oliveira and Pamela Zave. Vol. 2021. Lecture Notes in Computer Science. Springer, 2001, pp. 135–152. DOI: `10.1007/3-540-45251-6_8`.

[27] Michael Karr. “Affine relationships among variables of a program”. In: *Acta Informatica* 6.2 (1976), pp. 133–151. DOI: `10.1007/BF00268497`.

[28] Gary A. Kildall. “A unified approach to global program optimization”. In: *In Conference Record of the ACM Symposium on Principles of Programming Languages.* ACM Press, 1973, pp. 194–206. DOI: `10.1145/512927.512945`.

[29] Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. “Relational abstract interpretation for the verification of 2-hypersafety properties”. In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS’13, Berlin, Germany, November 4-8, 2013.* Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung. ACM, 2013, pp. 211–222. DOI: `10.1145/2508859.2516721`.

[30] Vincent Laviron and Francesco Logozzo. "SubPolyhedra: A (More) Scalable Approach to Infer Linear Inequalities". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Neil D. Jones and Markus Müller-Olm. Vol. 5403. Lecture Notes in Computer Science. Springer, 2009, pp. 229–244. DOI: 10.1007/978-3-540-93900-9_20.

[31] Thomas Lengauer and Robert Endre Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph". In: *ACM Transactions on Programming Languages and Systems* 1.1 (Jan. 1979), pp. 121–141. DOI: 10.1145/357062.357071.

[32] Heiko Mantel. "Preserving Information Flow Properties under Refinement". In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2001, pp. 78–91. DOI: 10.1109/SECPRI.2001.924289.

[33] Heiko Mantel and Henning Sudbrock. "Types vs. PDGs in Information Flow Analysis". In: *Logic-Based Program Synthesis and Transformation, 22nd International Symposium, LOPSTR 2012, Leuven, Belgium, September 18-20, 2012, Revised Selected Papers*. Ed. by Elvira Albert. Vol. 7844. Lecture Notes in Computer Science. Springer, 2012, pp. 106–121. DOI: 10.1007/978-3-642-38197-3_8.

[34] Antoine Miné. "A New Numerical Abstract Domain Based on Difference-Bound Matrices". In: *Proceedings of the Second Symposium on Programs As Data Objects*. PADO '01. London, UK, UK: Springer, 2001, pp. 155–172. DOI: 10.1007/3-540-44978-7_10.

[35] Antoine Miné. "A Few Graph-Based Relational Numerical Abstract Domains". In: *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*. Ed. by Manuel V. Hermenegildo and Germán Puebla. Vol. 2477. Lecture Notes in Computer Science. Springer, 2002, pp. 117–132. DOI: 10.1007/3-540-45789-5_11.

[36] Antoine Miné. "The Octagon Abstract Domain". In: *Higher Order Symbol. Comput.* 19.1 (Mar. 2006), pp. 31–100. DOI: 10.1007/s10990-006-8609-1.

[37] Christian Müller, Máté Kovács, and Helmut Seidl. "An Analysis of Universal Information Flow Based on Self-Composition". In: *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. Ed. by Cédric Fournet, Michael W. Hicks, and Luca Viganò. IEEE Computer Society, 2015, pp. 380–393. DOI: 10.1109/CSF.2015.33.

[38] Markus Müller-Olm and Helmut Seidl. "Analysis of Modular Arithmetic". In: *Programming Languages and Systems*. Ed. by Mooly Sagiv. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 46–60. DOI: `10.1007/978-3-540-31987-0_5`.

[39] Benedikt Nordhoff. "Information Flow Control via Dependency Tracking". In: *Archive of Formal Proofs* (Apr. 2021). `https://isa-afp.org/entries/IFC_Tracking.html`, Formal proof development.

[40] Torsten Robschink and Gregor Snelting. "Efficient path conditions in dependence graphs". In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida: ACM, 2002, pp. 478–488. DOI: `10.1145/581339.581398`.

[41] Enric Rodríguez-Carbonell and Deepak Kapur. "An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants". In: *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*. Ed. by Roberto Giacobazzi. Vol. 3148. Lecture Notes in Computer Science. Springer, 2004, pp. 280–295. DOI: `10.1007/978-3-540-27864-1_21`.

[42] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19. DOI: `10.1109/JSAC.2002.806121`.

[43] Axel Simon, Andy King, and Jacob M. Howe. "Two Variables per Linear Inequality as an Abstract Domain". English. In: *Logic Based Program Synthesis and Transformation*. Ed. by Michael Leuschel. Vol. 2664. Lecture Notes in Computer Science. Springer, 2003, pp. 71–89. DOI: `10.1007/3-540-45013-0_7`.

[44] Gregor Snelting. "Combining slicing and constraint solving for validation of measurement software". In: *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*. Ed. by Radhia Cousot and David A. Schmidt. Vol. 1145. Lecture Notes in Computer Science. Springer, 1996, pp. 332–348. DOI: `10.1007/3-540-61739-6_51`.

[45] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. "Checking Probabilistic Noninterference Using JOANA". In: *it - Information Technology* 56 (Nov. 2014), pp. 280–287. DOI: `10.1515/itit-2014-1051`.

[46] Mana Taghdiri, Gregor Snelting, and Carsten Sinz. “Information Flow Analysis via Path Condition Refinement”. In: *Proceedings of the 7th International conference on Formal aspects of security and trust*. FAST’10. Pisa, Italy: Springer, 2011, pp. 65–79. DOI: 10.1007/978-3-642-19751-2_5.

[47] Tachio Terauchi and Alex Aiken. “Secure information flow as a safety problem”. In: *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. Ed. by Chris Hankin and Igor Siveroni. Vol. 3672. Lecture Notes in Computer Science. Springer, 2005, pp. 352–367. DOI: 10.1007/11547662_24.

[48] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. “A sound type system for secure flow analysis”. In: *Journal of Computer Security* 4.2-3 (Jan. 1996), pp. 167–187. DOI: 10.3233/JCS-1996-42-304.

[49] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. “On PDG-based noninterference and its modular proof”. In: *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*. Ed. by Stephen Chong and David A. Naumann. ACM, 2009, pp. 31–44. DOI: 10.1145/1554339.1554345.

[50] Mark Weiser. “Program Slicing”. In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

[51] Bin Xin and Xiangyu Zhang. “Efficient Online Detection of Dynamic Control Dependence”. In: *Proceedings of the 2007 international symposium on Software testing and analysis*. ISSTA ’07. London, United Kingdom: ACM, 2007, pp. 185–195. DOI: 10.1145/1273463.1273489.