Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

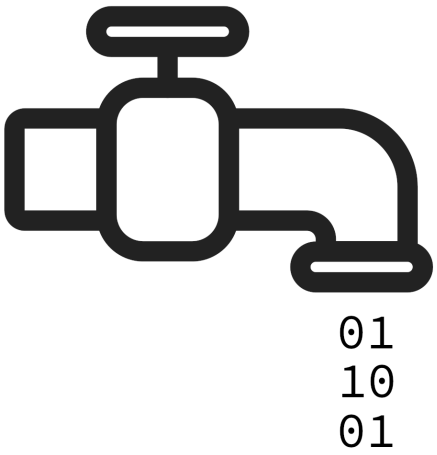
Lehrstuhl Prof. Dr.-Ing. Snelting

Bounding Information Leakage by Combining an Interpreter with Model Counting

Masterarbeit von

Tina Maria Strößner

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Johannes Bechberger
	M. Sc. Simon Bischof
Abgabedatum:	9. August 2021

Abstract

Information flow control (IFC) is the process of analyzing whether an attacker can gain information about sensitive data by observing the outputs of a program. The traditional approach to IFC is qualitative. The focus is on whether the program leaks *any* information. More relevant for practical purposes is the quantitative approach to IFC, where the aim is to measure *how much* information the program can leak.

Quantitative approaches to IFC are usually classed into one of two categories. Static analyses and dynamic analyses. We present an approach to quantitative information flow control that combines both. Our tool performs a SAT-based analysis of the program, that converts all information flows into propositional formulas. We combine this with a static quantitative IFC tool that is applied to parts of the program, where analyzing every program path is infeasible.

We show that the combination of the two approaches can improve the results of the quantitative analysis, compared to purely static or purely dynamic tools.

Zusammenfassung

Informationsflusskontrolle ist die Analyse von Programmen, mit dem Ziel zu überprüfen, ob es für einen Angreifer möglich ist, geheime Informationen zu erlangen, indem er die Ausgaben des Programms beobachtet. Traditionellerweise handelt es sich dabei um eine qualitative Analyse. Es wird bewertet *ob* Informationen zum Angreifer gelangen können. Für praktische Zwecke wichtiger ist allerdings die quantitative Informationsflusskontrolle. Hierbei wird gemessen, *wie viel* Information der Angreifer erlangen kann.

Verschiedene Ansätze zur quantitativen Informationsflusskontrolle werden üblicherweise in eine von zwei Kategorien eingeteilt: statische Analysen und dynamische Analysen. Wir präsentieren ein Verfahren zur quantitativen Informationsflusskontrolle, das Elemente aus statischen und dynamischen Analysen kombiniert. Wir wenden eine SAT-basierte Analyse auf das Eingabeprogramm an, die sämtliche Informationsflüsse in logische Formeln umwandelt. Dieser Ansatz wird kombiniert mit einer statischen quantitativen IFC Analyse, die auf Teile des Programms angewendet wird, wo das Analysieren eines jeden möglichen Ausführungspfades nicht effizient möglich ist.

Wir werden zeigen, dass die Kombination der beiden Ansätze die Ergebnisse der quantitativen Analyse verbessern kann im Vergleich zu Analysen, die rein statisch bzw. rein dynamisch arbeiten.

Contents

1. Introduction

1.1. Motivation

In February 2017, the American web services and web security company Cloudflare made headlines when a security bug was discovered in an HTML parser. A buffer overrun caused the servers to leak sensitive data, such as browser cookies, authentication tokens, or HTTP post bodies. The so-called "Cloudbleed" bug is one of many examples of software revealing sensitive data to unauthorized users. Such security threats have become more prominent, as more and more software is used in the handling of sensitive data [1, 2].

Qualitative Information Flow Control Qualitative information flow control aims to guarantee that a program's secret inputs do not influence its public outputs. Thus, a malicious attacker has no possibility of obtaining secret information simply by observing the program's outputs. This property is called *non-interference*.

For non-interferent programs, the secret inputs are guaranteed to not influence the public outputs. Unintentional data leaks are impossible. However, often this requirement is too strict for practical use. Consider the program shown in figure 1.1. While the secret password is not leaked in its entirety, an attacker can gather some information by observing whether their guess was correct. Hence, the non-interference property is violated. However, for most practical purposes, some information leakage is fully acceptable or even inevitable for the program's intended usage, like in the password checker example.

Quantitative Information Flow Control The desire to make information flow control applicable to more practical use cases gave rise to the notion of quantitative information flow control (QIFC). QIFC measures the amount of information leaked by a program in bits and, if required, compares it to a predetermined limit. The amount of leakage in the password checker example is one bit.

QIFC analyses are typically divided into two categories:

Static analyses are performed without executing the program and rely solely on examination of the source code. Such analyses usually deliver an upper bound for the amount of information a program might leak. However, this upper bound might be much larger than the actual leakage, due to the lack of knowledge about concrete control and data flows in the program.

Contrarily, *dynamic analyses* compute a program's leakage by simulating one or more program executions. For these executions, dynamic analyses might be able to deliver a

```
   Input password, guess: int
   Output match: int
1: if guess == password then
2:     match ← 1
3: else
4:     match ← 0
5: end if
```

Figure 1.1.: Program that checks if the public input `guess` matches the secret input `password` and reveals the result of the check to the user.

more precise estimate of the leakage than static analyses. However, these estimates are not guaranteed to be a sound upper bound, since it is often infeasible to analyze every possible execution of the program.

1.2. Related Work

While early mentions of quantifying information flow have been made, for example by Denning [3], a formal definition and theoretical groundwork for the QIFC problem are given more recently by Lowe [4] and Smith [5]. A recent publication by Alvim et al. [6] gives a comprehensive overview of the theory behind quantitative information flow control.

Numerous QIFC analyses based on model counting have previously been introduced.

Newsome, McCamant, and Song [7] transform their programs into boolean predicates that accurately model the programs' semantics. Using SAT-based techniques they then measure the program's channel capacity. The results are used to find false positives in a dynamic taint analysis to more accurately determine the amount of influence of an attacker on the program.

Klebanov, Manthey, and Muise [8] use the bounded model checker CBMC [9] to generate a boolean predicate and the d-DNNF-based #SAT tools SHARPSAT [10] and DSHARP [11] to find the number of models of these predicates. A similar approach is used by Biondi et al. in [12]. To address scalability issues with overly complex SAT formulas, they use an approximate model counter (in this case ApproxMC [13]). All these model-counting-based tools compute the channel capacity of the input program.

Chu and Hashimoto in [14] also combine the CBMC framework with an approximate model counter to estimate the pre-image size of a program input.

A static analysis based on constant bit analysis is presented in [15]. By analyzing bit-level dependencies on a PDG, the amount of leaked information can be estimated by calculating a minimal vertex cut.

A previous attempt at combining static and dynamic techniques has been made by McCamant and Ernst [16]. They transform programs into network graphs with edge

capacities corresponding to the amount of information that might flow between the corresponding program parts. Thus, the maximum leak of information corresponds to the maximum flow in the created network. They use this approach to estimate the leakage of the input program. However, their analysis is not sound and might underestimate leakage.

Dynamic techniques such as multi execution [17] or faceted values [18] offer the possibility of executing the program safely: Multiple executions are simulated simultaneously for different security levels. The true program outputs are only produced in the execution linked to their security. All other outputs are substituted by default values.

The analysis presented in [19] extends the dynamic *permissive upgrade strategy* to include a quantification of the information leakage to bound the amount of information leaked at run time.

1.3. Contributions and Overview

In this thesis, we define and compare two different measures for information leakage: the first is one based on min-entropy and is used by many of the QIFC tools mentioned above. The second one is based on the notion of dynamic leakage during a single program execution. We will present a model-counting-based analysis that can provide estimations for both quantities.

Additionally, we present an integration of static QIFC methods into our analysis to mitigate the difficulties that other model-counting-based tools have.

The analysis will be integrated into an interpreter that provides the possibility of executing a program while simultaneously providing the user with bounds on the information that is leaked by the program and the current execution.

This thesis will be structured as follows: We begin by introducing the theoretical concepts that our work is based on in section 2. In section 3, we introduce the basic design and theoretical foundations of the interpreter. The following section 4 extends the analysis to more advanced control flow structures.

In section 7 we benchmark our tool regarding aspects like precision and scalability and evaluate the results in comparison to other analyses.

2. Theoretical Background

This chapter presents the theoretical background of this thesis. We will explain the foundations of quantitative information flow control and the different measures used to quantify information leakage. Furthermore, we will outline important aspects of (approximate) model counting, as well as basic principles used in static (Q)IFC analyses *Nildumu* and *JOANA*, as both are used in our hybrid QIFC tool.

2.1. Quantitative Information Flow Control

Information flow control aims to guarantee the confidentiality of the secret input data of a program, by examining the flow of information through the program to public output channels, where the information becomes accessible to an attacker.

There are different ways in which such information leakage might happen: Explicit information flows are a consequence of data dependencies. They occur for example when a secret value is written to a public channel or the information is copied to another variable that is later leaked to a public channel. An example of a program is shown in figure 2.1. Implicit information flows are caused by control dependencies. Leaks through implicit flows happen when secret values affect the program's output by influencing the execution path. Figure 2.2 shows a program that contains an implicit flow. The program never assigns a secret value to an output variable. However, through the condition of the if-statement, the secret still influences the output value. Additional to explicit and implicit flows, an attacker could gain information through covert channels by observing a program's usage of different resources, such as time or memory [20].

The aim of qualitative information flow control is to prove the absence of explicit and implicit information flows, a property called *non-interference*. For real-world applications, leaking a certain amount of information is often required to build useful programs. In this case, the non-interference property is too strict. Instead, we wish to limit the amount of information that is leaked. Quantitative information flow control provides tools to measure how much information can be learned by an attacker about a program's secret inputs [5].

The central question of QIFC is: Given a program p that accepts some input `H` and produces some output `L`, how much information can an adversary $\mathscr{A}$ learn about `H` , if he knows the program p and observes `L`?

We will use the following assumptions for p and $\mathscr{A}$:

Input program We assume p to be a sequential, deterministic program that receives an input `H` and produces an output `L`. Both the input and output can be tuples that

```
   Input H : int
   Output L: int
1: L : int ← H % 10
```

Figure 2.1.: Example for information leakage through explicit flows, caused by a data dependency in line 1

```
   Input H : int
   Output L: int
1: L : int ← 0
2: if H == 42 then
3:     L ← 1
4: end if
```

Figure 2.2.: Example for information leakage through implicit flows, caused by a control dependency from line 2 to line 3

consist of multiple values. The concrete input and output values of a single execution are called h and l. The input h is an element of the set $\mathcal{H}$ of all possible inputs for a program. We assume that all program executions terminate.

Because the programs we consider are deterministic, each program p and input value h induces a mapping from the input value to the output value of the program. We write $[\![p]\!]_h(\mathtt{L})$ for the output value from the execution with h. $\mathcal{L}$ is the set of all possible outputs of the program. The set $\mathcal{L}$ is determined by p and H via the equation $\mathcal{L} := \{[\![p]\!]_h(\mathtt{L}) \mid h \in \mathcal{H}\}$

For a more detailed description of the input language we use, we refer to 3.1.

Security Lattice Input and output variables are associated with an element of a security lattice, describing its confidentiality level. We use a lattice with two elements: $\hat{l}$ for public values and $\hat{h}$ for secret values. If not otherwise specified, we consider all inputs to be secret and all outputs to be public.

Attacker Model We consider an adversary $\mathcal{A}$ that knows the source code of the program p and observes all outputs with a security level $\hat{l}$ after the execution has finished. The input value for an execution is chosen based on an underlying probability distribution. The distribution is known to the adversary.

We assume that the attacker is not able to extract any information through covert channels. The goal of the attacker is to guess the secret input of the program, using the information he can extract through observing the program's output.

Information Flow Measures Smith [5] characterizes information leakage with the following informal equation:

$$\text{Initial uncertainty} = \text{information leaked} + \text{remaining uncertainty} \tag{2.1}$$

In our scenario, the unknown value H is the initial uncertainty, measured by some entropy measure. The remaining uncertainty is the entropy of H after observing L.

Measuring Leakage with Vulnerability A widely used measure for information leakage is *min-entropy*, which is based on vulnerability. The vulnerability of a value X describes "the worst case probability that an adversary could guess the value of X in one try" [5]. If no other references are given, we follow the definitions from [5] in this section.

Definition 2.1.1 (Vulnerability and Min-Entropy)
Let X be a random variable and $\mathscr{X}$ the set of possible values for X. The *vulnerability* $V(X)$ is defined as

$$V(X) := \max_{x \in \mathscr{X}} P[X = x]$$

The min-entropy of X is given by

$$H_\infty(X) := \log_2 \tfrac{1}{V(X)}$$

Definition 2.1.2 (Conditional Entropy)
Given two random variables X and Y that are jointly distributed, the conditional entropy $H(X \mid Y)$ describes the uncertainty about X given Y. $H(X \mid Y)$ is defined as

$$H(X \mid Y) := \sum_{y \in \mathscr{Y}} P[Y = y] H(X \mid Y = y)$$

where

$$H(X|Y = y) := \sum_{x \in \mathscr{X}} P[X = x|Y = y] \log \frac{1}{P[X = x|Y = y]} \tag{2.2}$$

Using these definitions, Smith [5] proposes the following definitions for the leakage equation in 2.1:

The initial uncertainty is given by the min-entropy of the input value $H_\infty(\mathtt{H})$ and the remaining uncertainty is given by the conditional min-entropy $H_\infty(\mathtt{H}|\mathtt{L})$ of H after having observed L as the output. Thus the information leaked by a program is $H_\infty(\mathtt{H}) - H_\infty(\mathtt{H}|\mathtt{L})$. This quantity can be determined easily with the following definition and theorem:

Definition 2.1.3 (Channel Capacity)
Given a program p, the channel capacity of p is the logarithm of the number of distinct outputs than can be produced by p.

$$cc(p) := \log_2 |\mathscr{L}|$$

For deterministic programs, the channel capacity is the maximum entropy of L over all distributions of $\mathscr{H}$ [5].

Theorem 2.1 (Measuring Min-entropy through Channel Capacity)
For deterministic programs p, with $\mathscr{H}$ being uniformly distributed, the information leaked is equal to the channel capacity of p.

Measuring Leakage for Individual Program Executions While the vulnerability-based approach is appropriate to assess the risk of inadvertent information leakage for the program as a whole, it fails to give realistic measures in the dynamic case. If we consider the output of a single program run, the information an attacker may obtain from the output can be significantly greater than what the channel capacity of the program tells us [21].

We demonstrate this using the example program from 2.2. The possible outputs of the program are `0` and `1`, so the leakage, measured in bits, using the channel capacity formula is $\log_2 2 = 1$. Assuming integers are 64 bit wide, leaking a single bit seems acceptable in most cases.

Now, let's assume we run the program with the input $h := 42$. The output of the program will be 1. An attacker that observes this output and has access to the program code will now know that the secret input was 42. Instead of one single bit, the attacker has gained knowledge about all 64 bits of the secret input. For every other input, the attacker can only conclude that $\mathtt{H} \neq 42$. That leaves $2^{64} - 1$ possible inputs that are indistinguishable for the attacker.

This example shows that relying on min-entropy and channel capacity as a measure of information leakage can be dangerous if we care about the confidentiality of the secret inputs in every run of the program. Therefore, we now introduce measures that are more suited for the dynamic case.

We define an equivalence relation called the *indistinguishability relation* as shown in definition 2.1.4, where two inputs are related iff they produce the same program output. The equivalence class of this relation are pre-images of the possible program outputs (see definition 2.1.5). When the adversary observes the value l, he knows that the secret input must be an element of $\mathscr{H}_l$. Thus, the bigger the size of $\mathscr{H}_l$, the less likely the adversary is to guess the secret input in a single try [22, 5, 6].

Definition 2.1.4 (Indistinguishability Relation)
For each program p, we define the *indistinguishability relation* $\sim$ over $\mathscr{H}$ as:

$$\forall h, h' \in \mathscr{H} : h \sim h' \iff [\![p]\!]_h(\mathtt{L}) = [\![p]\!]_{h'}(\mathtt{L})$$

Definition 2.1.5 (Indistinguishability Set)
The indistinguishability set of a public output $l \in \mathscr{L}$ of a program p is given as:

observed output	channel capacity	dynamic leakage
$\mathtt{L} = 1$	2	64
$\mathtt{L} = 0$	2	$7.8 * 10^{-20}$

Figure 2.3.: Comparison of the two leakage measures for example 2.1. The quantities are measured in bit.

$$\mathscr{H}_l := \{h \in \mathscr{H} \mid [\![p]\!]_h(\mathtt{L}) = l\}$$

Consequently, in the dynamic case, the probability of an attacker guessing the secret in a single try depends on the size of the indistinguishability set for the output for the specific execution that is analyzed.

To reflect this notion in the leakage measure of our analysis, we redefine the *remaining uncertainty* in equation 2.1 as the conditional entropy of $\mathtt{H}$, given the outcome $\mathtt{L} = l$. The formula of this quantity is given in equation 2.2.

As mentioned above, the possible values for $\mathtt{H}$ are restricted to the elements of $\mathscr{H}_l$, so the probability $P[H = h|L = l] = 0$ for all $h \in \mathscr{H} \setminus \mathscr{H}_l$. Assuming a uniform distribution for the secret input values, we obtain the following leakage measure:

$$\begin{aligned} H(\mathtt{H}|\mathtt{L} = l) &:= \sum_{h \in \mathscr{H}} P[\mathtt{H} = h|\mathtt{L} = l] \log_2 \frac{1}{P[\mathtt{H} = h|\mathtt{L} = l]} \\ &= \sum_{h \in \mathscr{H}_l} \frac{1}{|\mathscr{H}_l|} \log_2 |\mathscr{H}_l| \\ &= \log_2 |\mathscr{H}_l| \end{aligned}$$

Theorem 2.2 (Dynamic leakage)
Assuming a uniform distribution of the inputs, the dynamic leakage of a single program run of a program p with a resulting output l is given by

$$\begin{aligned} L_{dyn}(p, l) &:= H_\infty(\mathtt{H}) - H(\mathtt{H}|\mathtt{L} = l) \\ &= \log_2 |\mathscr{H}| - \log_2 |\mathscr{H}_l| \end{aligned}$$

The table in figure 2.3 compares the two leakage measures defined in this section.

The relationship between dynamic leakage and min-entropy is shown in equation 2.3: The channel capacity of a program corresponds to the expected value of the dynamic leakage. The proof for the equation is given in the appendix B.

$$cc(p) = \mathbb{E}(L_{dyn}(p, l)) \tag{2.3}$$

In this thesis, we develop an analysis that can calculate both, the channel capacity of a program and the dynamic leakage of a single execution.

2.2. Program Representation

Control Flow Graph A control flow graph (CFG) is a program representation that highlights the control flows and possible execution paths in the program [23]. CFGs are widely used in compiler optimizations and static program analyses. The nodes of a CFG are a function's basic blocks, plus two special blocks `start` and `end`, that mark the single entry- and exit point of the function. An edge is inserted for every possible jump from one block to another. An example graph for the program from figure 2.2 can be seen in 2.4a.

We use BB_p for the set of all basic blocks of a program p and $b_1, b_2, \ldots$ for the blocks themselves. With exception of the `start`- and `end`-block, every block in the CFG has at minimum one predecessor and one successor.

Static Single Assignment Static single assignment (SSA) is a representation of the program, where every variable is assigned exactly once. If in the original program, a variable is written to more than once, a copy of the variable is created that replaces the original one from that point in the program [24].

To decide which copy of a variable reaches a statement that uses the value of that variable, ϕ-functions are used. ϕ-functions are placed at points in the program at which multiple control flow paths merge. In a CFG these points are basic blocks that have more than one predecessor. For each control flow path, the variable copy used on that path is given as an argument to the ϕ-function. During execution, the ϕ-function evaluates to the argument that belongs to the control flow path that was executed. The CFG in figure 2.4a contains a ϕ-function in block b_3. The value `L` defined in b_3 is assigned the value $\mathtt{L}_1$ if block b_2 is executed, otherwise $\mathtt{L}_0$ will be assigned to `L`.

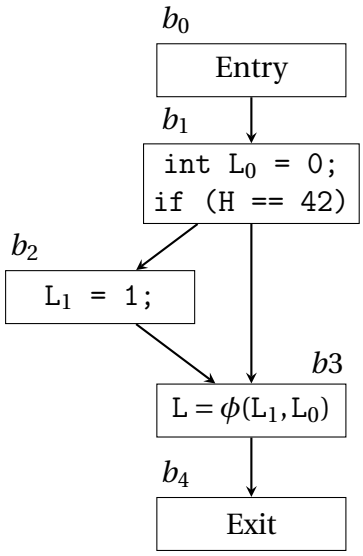
All data structures we use to represent the input program, are built using the SSA-form of the program.

Program Dependence Graph A program dependence graph (PDG) is an intermediate representation of a program that makes the program's data and control dependencies explicit. Its nodes are the program's operations and expressions and the edges represent the dependencies that exist between those [25]. The Program Dependence Graph of the program 2.2 is shown in figure 2.4b.

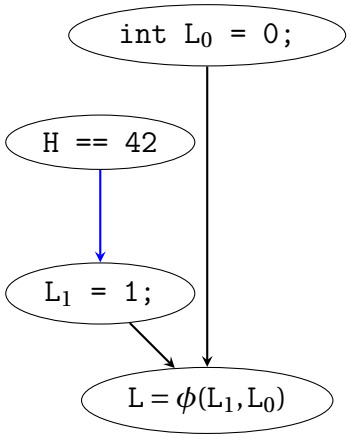
A data dependency edge from node x to node y exists, if the operation of the node y depends on a value that is defined in the node x.

A control dependency edge from node x to node y exists, if the outcome of x has an influence on whether node y will be executed.

Thus, a path $x \overset{*}{\rightsquigarrow} y$ between two nodes exists, if information might flow in the program from location x to location y. Consequently, if there is no path, no information can flow between the two statements. This property makes PDGs a popular data structure for information flow analysis [26, 27].



(a) Control Flow Graph



(b) Program Dependence Graph
The black edges are data dependencies, while blue edges show control dependencies

Figure 2.4.: Different graph representations for the program in 2.2

2.3. Program Slicing

Slicing is a technique used to find those sections of a program that are influenced by (forward slice) or influence (backward slice) a given location in the program [28]. This location is called the *slicing criterion* and is a tuple $\langle s, v \rangle$ of a program statement s and a variable v. The *backward slice* with respect to the criterion $\langle s, v \rangle$ contains all statements that influence the value of v at point s in the program. A *forward slice* for the criterion $\langle s, v \rangle$ contains all statements that are influenced by the value of v assigned at point s in the program.

Computing a program slice can be efficiently done using the PDG of the given program. Since all dependencies are explicitly represented as edges, the computation of a program slice is reduced to a reachability problem [29]:

The backward slice for a node v is given as the set of all nodes v', for which a path $v' \overset{*}{\rightsquigarrow} v$ exists. The forward slice of node v is given as the set of all nodes v', for which a path $v \overset{*}{\rightsquigarrow} v'$ exists.

Horwitz [30] noted that interprocedural slices could be computed similarly using system dependence graphs.

Tools like JOANA use slicing techniques on system dependence graphs to analyze the information flow in programs and give non-interference guarantees where possible [31].

```
Input H : int
Output sum: int

i : int ← 0
sum : int ← 0
product : int ← 1
while i ≤ H do
    sum ← sum + i
    product ← product ∗ i
    i++
end while
return
```

```
Input H : int
Output sum: int

i : int ← 0
sum : int ← 0

while i ≤ H do
    sum ← sum + i

    i++
end while
return
```

Figure 2.5.: The right side shows a backward slice of the function on the left for the slicing criterion ⟨return, sum⟩

2.4. Model Counting

Given a propositional formula F, the model counting problem (#SAT) is the problem of finding the number of distinct variable assignments for F, for which F evaluates to `true` [32]. So the solution for the formula shown in 2.6 is #F = 3, with the fulfilling variable assignments being $\{x = \texttt{true}, y = \texttt{false}\}$, $\{x = \texttt{true}, y = \texttt{true}\}$ and $\{x = \texttt{false}, y = \texttt{false}\}$. We will denote the model count of a propositional formula f as $mc(f)$.

#SAT is a generalization of the SAT problem and falls into the #P-complete complexity class, as demonstrated by Valiant in [33].

Early exact model counting techniques, such as [34], or the well-known tool sharpSAT [10] use a DPLL-style exploration of the solution space. Another class of model counters instead employ complex transformations to turn the given CNF formula into a different representation, which makes model counting a far easier problem. Common are transformations into *binary decision diagrams* [35] or *deterministic, decomposable negation normal form* [36]. An example of such a model counter is the `c2d` tool by Darwiche [37].

Approximate Model Counting State-of-the-art exact model counters scale to a couple of hundred variables. This limit can be pushed to around 1,000 variables if we allow approximate solutions [32]. The first approximate #SAT algorithm for DNF formulas was introduced by Luby and Karp in [38] used Monte-Carlo techniques. This approach was extended to work on CNF formulas by Chakraborty, Meel and Vardi in [13]. Both procedures fall under category of (ϵ, δ)-counters: for $0 < \epsilon, \delta \leq 1$, the approximated solution $\#F_{approx}$ to the true solution of the problem #F, lies in the interval $[(1+\epsilon)^{-1}\#F, (1+\epsilon)\#F]$ with a probability of $1-\delta$ [38, 13].

$$F := x \vee \neg y$$

Figure 2.6.: Propositional formula with 2 variables

Projected Model Counting Given a set of propositional variables $\mathcal{V}$ and a propositional formula F over $\mathcal{V}$, projected model counting (#∃SAT) is the problem of finding the number of assignments to a set of priority variables $\mathscr{P} \subseteq \mathcal{V}$, such that the assignment can be extended to an assignment over $\mathcal{V}$ that fulfills F. Considering the example from 2.6 and a priority set $\mathscr{P} = \{x\}$, the number of projected models is 2, with both possible assignments of x being extendable to a fulfilling assignment over all variables by setting $y = \texttt{false}$. We will denote the projected model count of a propositional formula f over the set of priority variables $\mathscr{P}$ as $mc_{\mathscr{P}}(f)$.

The problem is introduced by Aziz et al. in [39]. Solving projected model counting is usually done using modified algorithms for the #SAT problem. Aziz et al. introduce approaches based on the DPLL strategy, where search decisions are first made on non-priority variables, and on the d-DNNF strategy, where non-priority variables are "forgotten" by replacing them with the value `true`.

3. Basic Analysis Design

The analysis we will present in this thesis combines static and dynamic approaches to the QIFC problem. It runs in several stages:

First is a static pre-processing stage. It identifies program parts that are critical to the flow of information and restricts the subsequent analysis to those parts.

Following the pre-processing, a dependency analysis examines possible program executions and evaluates the explicit and implicit information flow along these execution paths. We encode these information flows in propositional formulas and evaluate them using an approximate model counter. The generated boolean predicates can be used to estimate both the channel capacity of the input program as well as the dynamic leakage for a specific execution.

During the evaluation of the channel capacity, the generated boolean predicates might prove to be too complex to be evaluated by a model counter. In this case, our tool will split the program into segments. The segments are analyzed separately and the results are combined for an overall estimation of the program's channel capacity. The analysis of the segments will either be done using the previously generated boolean formulas or by the static QIFC tool Nildumu.

The analysis is integrated into an interpreter that will execute the program for a given input and, additionally to the channel capacity, will give an estimation for the dynamic leakage of the execution.

This chapter will focus on the basic principles of the dependency analysis: The generation of the boolean predicates and the relation between those predicates and the amount of information leaked by the program. For this, we assume that input programs have no loops and no function calls. How these structures can be handled is discussed in section 4.

3.1. Input Programs

Input programs are written in a variant of the `while`-language with functions. The language contains the following control structures, using their standard semantics:

- sequential composition
- assignments
- `if`-statements
- `while`-statements
- `break`-statements
- (recursive) function calls

The right-hand side of an assignment is an expression that uses the standard arithmetic and bitwise boolean operators. Boolean expressions used in `while`- and `if`-

statements are defined in the standard way. The output value of the program is revealed at the end of the execution. In particular, we assume that the leak of the value happens outside any control flow structures, such as loops or branches of of-statements.

As already mentioned, we disallow loops and function calls in our input programs for now.

We continue to use the notations introduced in section 2.1 for the input program. Furthermore, we denote the set of all statements (expressions) that are part of the language as `Stmts` (`Expr`). Subscripts, such as $\texttt{Stmts}_q$ ($\texttt{Expr}_q$) indicate the set of statements (expressions) that belong to the program part q, where q could be a loop, a function, or the program as a whole.

In our analysis, we work with the input's CFG as well as the PDG, both in SSA form. To further navigate the data structures, we use the following definitions:

Definition 3.1.1 (CFG Predecessors and Successors)
The functions will return the set of predecessor and successor blocks respectively for the given basic block.

$$pred\colon \mathrm{BB}_p \to 2^{\mathrm{BB}_p}$$
$$succ\colon \mathrm{BB}_p \to 2^{\mathrm{BB}_p}$$

We assume for $pred(b)$ that the returned set of predecessors for b is ordered and that the order corresponds to the arguments of any ϕ-functions that might be present in b.

In our analysis, we view all values as bit vectors. All values are signed integers of a fixed width w. We represent the numerical value of an integer as a bit vector using the following map:

Definition 3.1.2 (Bit Vector)
The function

$$bv\colon \mathbb{Z} \to \{0,1\}^w$$

maps integers to bit vectors of length w, where $bv(n)$ is the two's complement representation of the integer n. The returned value $bv(n)$ is subject to possible over- or underflows, should the number n not be representable as a w-bit two's complement number.

Execution Value We alter the definition of $[\![p]\!]_h : \{L\} \to \mathscr{L}$ to a function

$$[\![p]\!]_h\colon \mathrm{VAL}_p \to \{0,1\}^w \cup \{\bot\}$$

that takes a program value as an argument and returns the numerical value that was assigned during the execution with input h. The numerical value is given as a bit vector of the number's two's complement. If in this execution, the value remains undefined, because the corresponding assignment instruction wasn't executed, the function will return $\bot$. Because the program p does not contain loops or function calls, no assignment can be executed more than once. Thus, $[\![p]\!].(\cdot)$ is well-defined.

Input $\mathtt{H} := (\mathtt{H}^2\mathtt{H}^1\mathtt{H}^0)$: int
Output $\mathtt{L}_1$: int
1: $\mathtt{L}_0 \leftarrow \mathtt{H} \,\&\, 110_2$ $[H^2\ H^1\ 0]$
2: $\mathtt{L}_1 \leftarrow \mathtt{L}_0 \mid 010_2$ $[H^2\ 1\ 0]$

Figure 3.1.: Introductory example program. Behind each line of code, we have written the propositional vector that represents the assigned value

3.2. Foundations of Dependency Analysis

The aim of the dependency analysis is to generate a vector of propositional formulas for each program value that we call the *dependency vector*. This vector contains a formula for each bit of the value, that encodes the state of the bit depending on the bits of the input value. More specifically, the formula contains variables that represent the bits of the input value and if we initialize these variables with the bits of an input, the dependency vector of a value will evaluate to its execution value.

Introductory Example Before giving a detailed explanation of our analysis, we will demonstrate the basic principles in a short example. The program we want to analyze is shown in figure 3.1. The program takes an input value, performs two bitwise operations on the value, and then leaks the result.

Analyzing the program line by line, we can formulate conditions for the assigned values that depend on the values and operations used in the assignment expression:

- On line 1, the program assigns to the value L_0 the result of a bitwise `&`-operation. The right-hand operand is a constant. Because the least significant bit of this constant is `0`, the least significant bit of L_0 must also be `0`. The two left-hand bits of L_0 are identical to the corresponding bits in `H` . Thus, we can describe the value L_0 by the vector $[H^2\ H^2\ 0]$.
- On line 2, using the same approach as above, we can ascribe to the value L_1 the vector $[H^2\ 1\ 0]$

The dependency vector $[H^2\ 1\ 0]$ which we computed for the output value $\mathtt{L}_1$, describes all possible output values of the example program. The vector shows that the last two bit of the output are constant and that the most significant bit is equal to the most significant bit of the input. From this information, we can conclude the following:

- The *channel capacity* of the example program is $\log_2(2) = 1$, because the program has only two possible outputs: 110_2 iff the most significant bit of `H` is `1` and 010_2 iff the most significant bit of `H` is `0`.
- Given the value of $\mathtt{L}_1$ after an execution, we can infer the value of the most significant bit of `H`. We have no information about the other two bits. For any pos-

sible output $l = (l^2 l^1 l^0)$, its indistinguishability set is given by $\mathscr{H}_l := \{(h^2 h^1 h^0) \in \{0,1\}^3 \mid l^2 = h^2\}$. The *dynamic leakage* of a single execution is $\log_2(2^3) - \log_2(2^2) = 1$.

Basic Definitions and Notation Propositional formulas are made up of boolean constants $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$, boolean variables $b_i \in \text{VAR}_{\mathbb{B}}$ and the standard boolean operators $\{\neg, \wedge, \vee, \Longrightarrow, \Longleftrightarrow\}$. $\mathscr{F}$ is the set of all boolean formulas over $\text{VAR}_{\mathbb{B}}$.

In order to relate bit-vectors and propositional formulas to each other, we use the following definitions:

Definition 3.2.1 (Mapping Bits to Boolean Constants)
The bijective map $\mathscr{B} : \{0,1\} \rightarrow \mathscr{F}$ maps a bit to a boolean constant.

$$\mathscr{B} : \{0,1\} \rightarrow \mathscr{F}$$

$$0 \mapsto \texttt{false}$$
$$1 \mapsto \texttt{true}$$

Throughout this thesis, we will apply $\mathscr{B}(\cdot)$ and $\mathscr{B}^{-1}(\cdot)$ implicitly where needed.

Definition 3.2.2 (Mapping Values to Vectors of Boolean Variables)
The map

$$\mathscr{V}ar : \text{VAL}_p \rightarrow \mathscr{F}^w$$

assigns a vector of fresh propositional variables to a program value. Each boolean variable is used to represent a bit of the value. A boolean variable is fresh if it is not yet used to represent any other bit. This means that for values $v_0 \neq v_1 \in \text{VAL}_p$ we have $\mathscr{V}ar(v_0) \cap \mathscr{V}ar(v_1) = \emptyset$.

We will later use the function $\mathscr{V}ar(\cdot)$ to instantiate propositional variable vectors that represent the input value(s) of the program we are analyzing. The dependency vectors we compute are based on the variable set of these variable vectors. They form the so-called *independent set*:

Definition 3.2.3 (Independent Set)
Let $\{\texttt{H}_0, ..., \texttt{H}_m\}$ be the set of input values of p. The set of boolean variables

$$\mathscr{V}ar_p := \bigcup_{i \in \{1,...,m\}} \mathscr{V}ar(\texttt{H}_i)$$

is called the independent set. The independent set defines the variables that are used in propositional formulas during the analysis.

Dependency Analysis for Explicit Information Flow In the introductory example, we introduced the idea of the *dependency vector*: A vector of propositional formulas that represents the state of the value's bits in terms of the bits of the input value. In this section, we will explain how the dependency vector can be computed.

Definition 3.2.4 (Expression Evaluation and Dependency Vectors)
The function

$$\mathscr{E} : \texttt{Expr} \to \mathscr{F}^w$$

evaluates program expressions and computes a vector of propositional formulas that represent the expression result. The computation of $\mathscr{E}(e)$ is shown in figure 3.2.

Definition 3.2.5 (Dependency Vector)
The *dependency vector* function maps each value to a vector of propositional formulas. For a value v that is defined in a statement as $v := e$, we define:

$$dVec : \textsc{Val}_p \to \mathscr{F}^w,$$
$$dVec(v) := \mathscr{E}(e)$$

The map is well-defined since p is given in SSA-form which means that each value is defined exactly once. We use $dVec(v)^i$ to mean the i-th entry of the vector $dVec(v)$.

To assign each value its dependency vector, we compute $\mathscr{E}(e)$, for the expression e that defines the value in question. For accesses to input variables and constants, the evaluation of $\mathscr{E}(\cdot)$ is straightforward. For other expressions, the evaluation of $\mathscr{E}(e)$ usually depends on the dependency vectors of use-values of e. We can guarantee that the needed dependency values have previously been computed if we evaluate the dependency vectors of program values in the order in which the values are defined in the program.

3.3. Measuring Information Leakage with Dependency Vectors

The entries of a value's dependency vector describe on a bit level, which execution value will be assigned to the value for a certain input value. They take into account all data dependencies in the program. Therefore, we can capture all explicit information flows using the dependency vectors.

To demonstrate the connection between a value's dependency vector and its execution value for a particular program run, we evaluate the propositional formulas using the following truth assignment:

Definition 3.3.1 (Evaluation of Dependency Vectors)
Every concrete input $h \in \mathscr{H}$ for the input value H induces a truth assignment: Given the concrete input value h for the input variable H, we assign the boolean variables in $\mathscr{V}ar(\texttt{H})$ the values of the bits in h. The evaluation of a propositional formula f with respect to the truth assignment induced by the input h will be denoted by $\mathscr{V}_h(f)$.

$$\text{Let } e := n, \quad n \in \mathbb{Z}$$
$$\mathscr{E}(e) := \mathscr{B}(bv(n))$$

(a) Constant Values are represented by dependency vectors that contain boolean constants. The vector entries correspond to the two's complement representation of the constant's numerical value.

$$\text{Let } e := \mathtt{H}$$
$$\mathscr{E}(e) := \mathscr{V}ar(\mathtt{H})$$

(b) Input parameters are represented by a vector of boolean variables. The variables are part of the independent set $\mathscr{V}ar_p$ of p and dependency vectors of non-input variables are defined as a function of these variables.

$$\text{Let } e := v, \quad v \in \text{VAL}_p$$
$$\mathscr{E}(e) := dVec(v)$$

(c) Variable accesses are evaluated to the dependency vector of the accessed variable.

$$\text{Let } e := e_0 \oplus e_1 \text{ or } e := \oplus e_0$$

(d) Dependency vectors of unary or binary arithmetic expressions and unary or binary bitwise boolean expressions are computed using the combinatorial logic of the two's complement. Even though this logic normally operates on bits (either `0` or `1`), we can apply the boolean operations also to propositional formulas, without changing their semantics.

Figure 3.2.: Definition of $\mathscr{E}(e)$ for different types of expressions e

Using the truth assignment induced by a particular input $h \in \mathscr{H}$, the dependency vectors fulfill the following theorem:

Theorem 3.1 (Equivalence Theorem)
Given an input value $h \in \mathscr{H}$ for the program p, let v be an arbitrary value in the program p for which $[\![p]\!]_h(v) \neq \bot$. The relation between the dependency vector and the execution value of v is given by:

$$\forall 0 \leq i < w : \mathscr{V}_h(dVec(v)^i) \iff [\![p]\!]_h(v)^i$$

At this stage, we have not yet introduced the analysis of programs with diverging control flow. For now, the theorem can only be applied to programs with linear control flow. A proof for the theorem is given in appendix B.

Using theorem 3.1, we can compute both leakage measures with the following lemmata:

Lemma 3.2 (Dynamic Leakage)
Let p be a program with the concrete value l of the output variable L being leaked to a public output channel during the execution of p with input h. The size of l's indistinguishability set $\mathscr{H}_l$ is given by the number of models of the formula:

$$F_{dyn} : \bigwedge_{0 \leq i < w} \left([\![p]\!]_h(\mathtt{L})^i \iff \mathscr{V}_h(dVec(\mathtt{L}))^i \right)$$

The formula F_{dyn} contains only the boolean variables from the set $\mathscr{V}ar_p$. Each model of F_{dyn} thus is a truth assignment $\beta : \mathscr{V}ar_p \to \mathbb{B}$. If for a truth assignment β the formula F_{dyn} is fulfilled, the dependency vector of the output, evaluated with respect to this truth assignment, is equivalent to the bit vector l. From theorem3.1, it follows that the execution of p with the input h induced by β will result in the output l.

Assuming a uniform distribution of $\mathscr{H}$, the dynamic leakage of the execution of p with the input h is given by

$$L_{dyn}(p,h) = \log_2(|\mathscr{H}|) - \log_2(mc(F_{dyn}))$$

Lemma 3.3 (Channel Capacity)
Let p be a program with an output value L. Let $o := [o_0 ... o_w]$ be a vector of boolean variables with $o \cap \mathscr{V}ar_p = \emptyset$.

The number of distinct outputs of p is given by the projected model count of the formula

$$F_{cc} : \bigwedge_{0 \leq i < w} \left(o_i \iff dVec(\mathtt{L})^i \right)$$

with the variables in $o = [o_0, ..., o_w]$ as the set of priority variables. Assuming a uniform distribution of $\mathscr{H}$, the channel capacity is given by

$$cc(p) = \log_2(mc_o(F_{cc}))$$

```
    Input H : int
    Output L: int
 1: if H < 0 then
 2:     L ← 0
 3: else
 4:     H ← H − 1
 5:     if H < 0 then
 6:         L ← 1
 7:     else
 8:         L ← 2
 9:     end if
10: end if
```

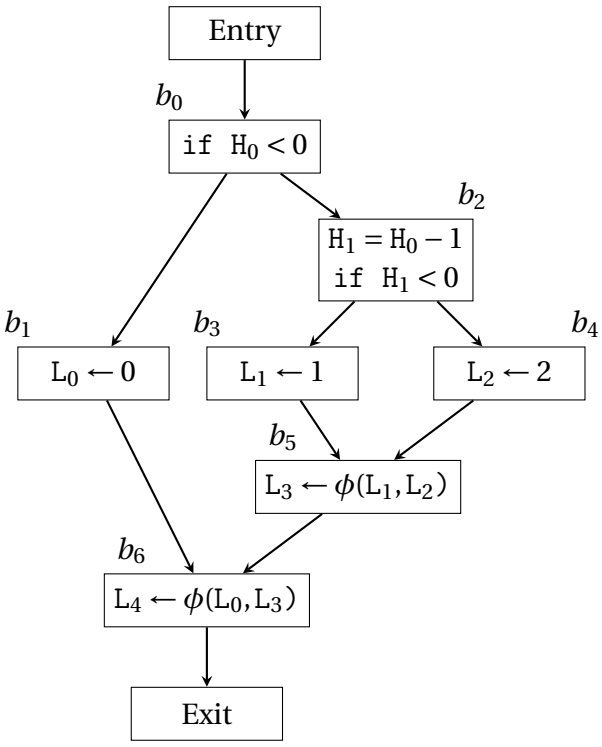


Figure 3.3.: Program text and CFG of a short example program. The program returns 0 if $H < 0$, 1 if $H == 0$ and 2 otherwise. The CFG is in SSA-form.

3.4. Dependency Analysis for Implicit Information Flow

Implicit information flow occurs when an attacker can draw conclusions about the secret inputs by observing the values of the public outputs and then reconstructing the execution path of a program execution. In this section we will extend the dependency analysis from before to include implicit information flows caused by `if`-statements. Implicit information flows from more complex control flow structures, such as loops and function calls, are discussed in chapter 4. Throughout this section, we use the program shown in figure 3.3 as an example to demonstrate the individual steps of the analysis.

To include implicit information flow in the analysis, we develop the function $exec$: $\mathrm{BB}_p \rightarrow \mathscr{F}$ that assigns each basic block b of a program a propositional formula $exec(b)$ that is fulfilled by the inputs iff the block b is executed.

In a CFG, the edges represent the jumps between basic blocks. We define the edge condition function $follow : \mathrm{BB}_p \times \mathrm{BB}_p \rightarrow \mathscr{F}$ to annotate CFG edges with expressions that describe when the jump between the two blocks is taken during an execution. The edge annotations are given as propositional formulas, that already take into account the dependency vectors that were computed for the operands. Figure 3.4 shows the CFG of the example program 3.3 with edges being annotated with their edge conditions.

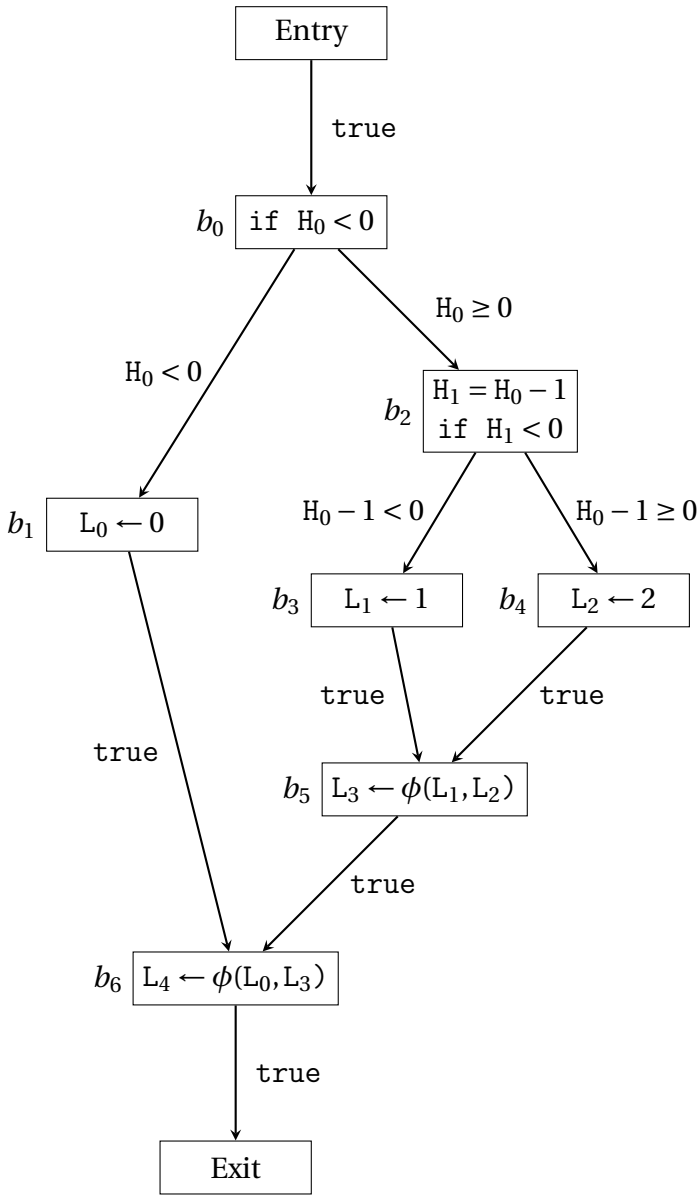


Figure 3.4.: CFG from figure 3.4 with annotated edges. The annotations represent the formulas $follow(e)$. For easier reading the formulas are given as propositional formulas with linear integer arithmetic instead of bit vector logic.

Execution Conditions for Basic Blocks A basic block b in a program is executed, iff one of its predecessor blocks is executed and the condition for execution to jump from said predecessor to the block b is fulfilled. A special case is a program's entry block, which is executed in every run. This leads to the following definition:

Definition 3.4.1 (Execution Condition)
For every basic block $b \in \mathrm{BB}_p$, we define its execution condition as:

$$exec\colon \quad BB_p \to \quad \mathscr{F}$$
$$b \mapsto \begin{cases} \texttt{true} & b = \texttt{entry} \\ \bigvee\limits_{p \in pred(b)} \big(follow(p,b) \wedge exec(p)\big) & \text{otherwise} \end{cases}$$

Lemma 3.4 (Correctness of $exec(\cdot)$)
For every basic block $b \in \mathrm{BB}_p$ and its execution condition $exec(b)$,
$\mathcal{V}_h(exec(b)) = \texttt{true} \iff$ basic block b is executed in a program run with input h.

The proof for the lemma is given in appendix B in conjunction with the proof for theorem 3.1.

Example Computation Table 3.5 shows the execution conditions of all the basic blocks of example 3.3. We show how these results were computed in detail for blocks b_3 and b_5:

- Basic block b_3 has one predecessor b_2 with $exec(b_2) = \mathtt{H}_0 \geq 0$. The edge (b_2, b_3) represents a conditional jump that is taken iff the condition $follow((b_2,b_3)) := \mathtt{H}_0 - 1 < 0$ is fulfilled.

$$\begin{aligned} exec(b_3) &:= follow((b_2,b_3)) && \wedge && exec(b_2) \\ &= \mathtt{H}_0 \geq 0 && \wedge && \mathtt{H}_0 - 1 < 0 \end{aligned}$$

- Basic block b_5 has two predecessors b_3 and b_4. Their execution conditions are given in table 3.5. Both incoming edges of b_5 represent unconditional jumps, thus their edge conditions evaluate to `true`.

$$\begin{aligned} exec(b_5) &:= follow((b_3,b_5)) \wedge exec(b_3) && \vee && follow((b_4,b_5)) \wedge exec(b_4) \\ &= \texttt{true} \wedge (\mathtt{H}_0 \geq 0 \wedge \mathtt{H}_0 - 1 < 0) && \vee && \texttt{true} \wedge (\mathtt{H}_0 \geq 0 \wedge \mathtt{H}_0 - 1 \geq 0) \\ &= \mathtt{H}_0 \geq 0 \wedge (\mathtt{H}_0 - 1 < 0 \vee \mathtt{H}_0 - 1 \geq 0) \\ &= \mathtt{H}_0 \geq 0 \end{aligned}$$

b	$exec(b)$
entry	true
b_0	true
b_1	$\mathtt{H}_0 < 0$
b_2	$\mathtt{H}_0 \geq 0$
b_3	$\mathtt{H}_0 \geq 0 \wedge \mathtt{H}_0 - 1 < 0$
b_4	$\mathtt{H}_0 \geq 0 \wedge \mathtt{H}_0 - 1 \geq 0$
b_5	$\mathtt{H}_0 \geq 0$
b_6	true
exit	true

Figure 3.5.: Evaluation of $exec(\cdot)$ for the basic blocks of program 3.3

Combining Implicit and Explicit Information Flows in ϕ-functions The execution conditions introduced in the previous section can be used to evaluate which basic blocks will be executed depending on the inputs. They contain all control flow dependencies that are present in the program. The formulas describing the implicit flows are integrated into the dependency vectors when a value is assigned the result of a ϕ-function. Let the value v_2 be defined via $v_2 := \phi(v_0, v_1)$, $v_0, v_1 \in \mathrm{VAL}_p$. The assignment expression is part of basic block b_2 with predecessors $\{b_0, b_1\}$. The situation is shown in figure 3.6a.

To evaluate the ϕ-expression and compute the dependency vector for value v_2, we use the ternary operator $\mathbb{IF}(\cdot,\cdot,\cdot)$:

Definition 3.4.2 (Ternary Operator)
We define the ternary operator $\mathbb{IF}(\cdot,\cdot,\cdot)$ as:

$$\mathbb{IF} : \mathscr{F} \times \mathscr{F} \times \mathscr{F} \to \mathscr{F}$$
$$\mathbb{IF}(c, x, y) := (c \implies x) \wedge (\neg c \implies y)$$

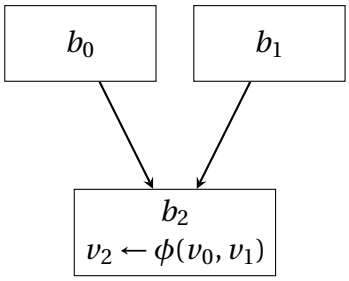
We canonically extend the definition to include propositional vectors:

$$\mathbb{IF} : \mathscr{F} \times \mathscr{F}^k \times \mathscr{F}^k \to \mathscr{F}^k$$
$$\mathbb{IF}(c, x, y) := [\mathbb{IF}(c, x^i, y^i)]_{i=0}^k$$

The dependency vector of the value v_2 can then be computed using the definition of $\mathscr{E}(\cdot)$ for ϕ-functions given in figure 3.6b. The correctness of this definition follows from the following considerations:

If we compute the dependency vector $dVec(v_2) := \mathscr{E}(\phi(v_0, v_1))$ of the value v_2, we can assume that the basic block b_2 is executed. Otherwise, any assignment to v_2 and the information contained in the assignment is irrelevant for the leakage analysis of the execution in question.

If the basic block b_2 is executed, at least one of its predecessors b_0, b_1 must have been executed as well for the execution to reach b_2. Furthermore, it is impossible for both of b_2's predecessors to have been executed since so far we only consider loop-free programs, which have no back-edges in their CFG. It follows that the condition



(a) Section of a CFG that contains a ϕ-function.

$$\mathscr{E}(\phi(v_0, v_1)) :=$$
$$\mathbb{IF}(exec(b_0), dVec(v_0), dVec(v_1))$$

(b) Extension of the definition of $\mathscr{E}(\cdot)$ in 3.2 for the evaluation of ϕ-functions.

Figure 3.6.: Handling of ϕ-functions during the dependency analysis

$exec(b_0) \veebar exec(b_1)$ must hold for any input value. It is therefore sufficient to check the execution condition of b_0 in the definition of $\mathscr{E}(\phi(v_0, v_1))$.

Example (cont'd) We complete the dependency analysis of the example program 3.3 using the algorithm shown in figure 3.7. The algorithm returns the dependency vector of the leaked value. Figure 3.8 shows the dependency vectors of all program values. In SSA-form, $\mathtt{L}_4$ is the value that corresponds to the program's output. The algorithm returns:

$$\begin{aligned} dVec(\mathtt{L}_4) &= \mathbb{IF}(\mathtt{H}_0 < 0, dVec(\mathtt{L}_0), dVec(\mathtt{L}_3)) \\ &= \mathbb{IF}(\mathtt{H}_0 < 0, 0, \mathbb{IF}(\mathtt{H}_0 \geq 0 \wedge (\mathtt{H}_0 - 1) < 0, dVec(\mathtt{L}_1), dVec(\mathtt{L}_2))) \\ &= \mathbb{IF}(\mathtt{H}_0 < 0, 0, \mathbb{IF}(\mathtt{H}_0 \geq 0 \wedge (\mathtt{H}_0 - 1) < 0, 1, 2)) \end{aligned}$$

Algorithm 3.1 Dependency Analysis

Input $CFG(p) = (\mathrm{BB}_p, E)$: CFG of input program p in SSA form.
Output $leaked : \mathscr{F}^w$

1: $blocks : (\mathrm{BB}_p \to \mathscr{F})$
2: $dVec : (\mathrm{VAL}_p \to \mathscr{F}^w)$
3: **for** $b \in \mathrm{BB}_p$ in topological order **do**
4: $\quad blocks(b) \leftarrow exec(b)$
5: $\quad$ **for** $v := e \in Statements(b)$ **do**
6: $\quad\quad dVec(v) \leftarrow \mathscr{E}(e)$
7: $\quad$ **end for**
8: **end for**
9: $leaked \leftarrow dVec(\mathrm{L})$

Figure 3.7.: Algorithm for the dependency analysis of call-free, loop-free programs

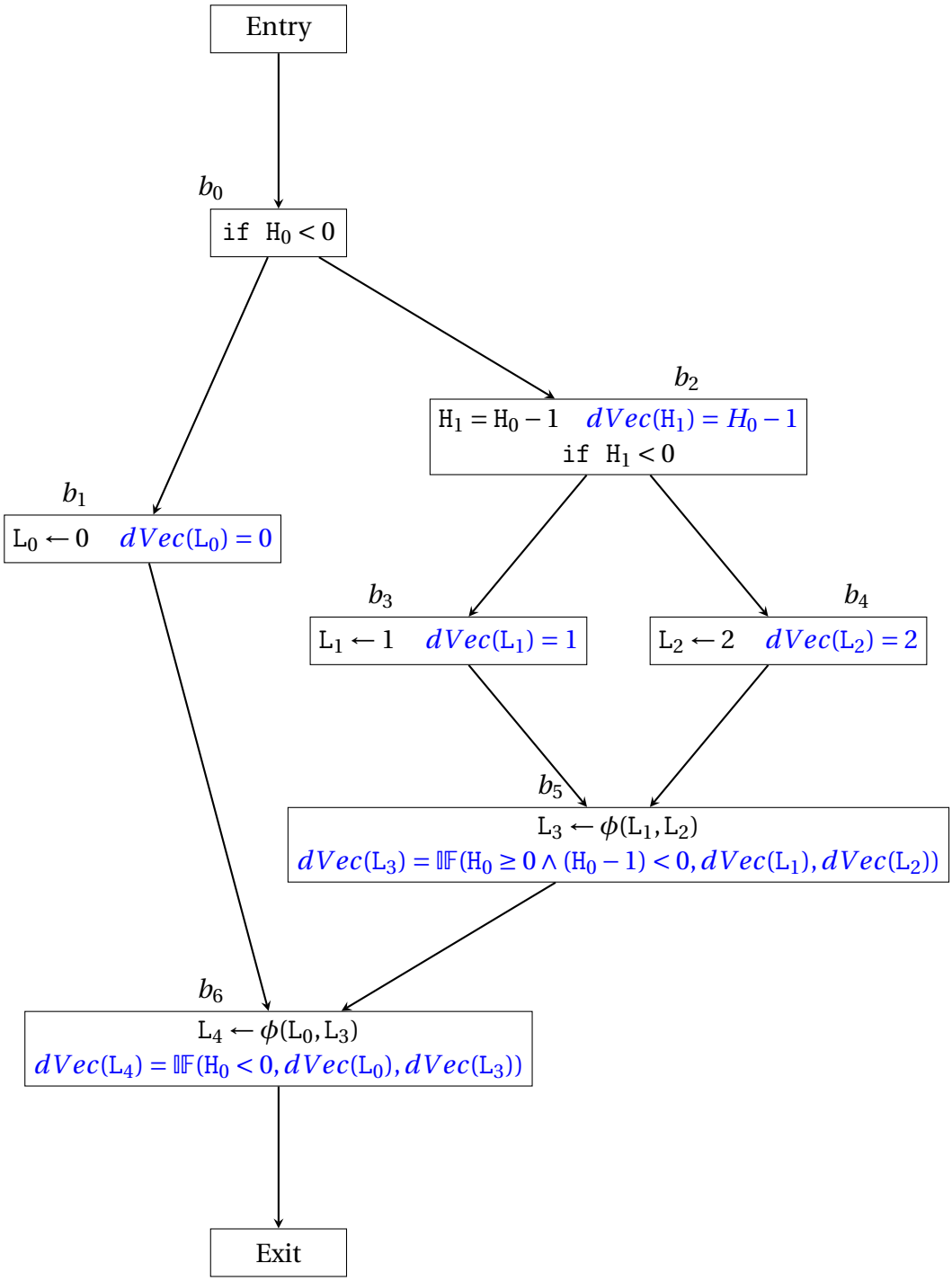


Figure 3.8.: CFG of program 3.3 annotated with the dependency vectors of each program value

4. Extended Dependency Analysis

Chapter 3 introduced the basic principles of the dependency analysis and how the results of the dependency analysis can be used to measure the amount of leaked information for a given program. In this chapter, we will extend the dependency analysis to include the handling of arrays and more complex control flow structures, such as loops and function calls.

Sequential Execution Value So far, we used the execution value $[\![p]\!]_h(v)$ to get the actual value of v during the execution with input h. Allowing loops and function calls in our input programs means that it is possible for statements to be executed multiple times. A value v can thereby have more than one execution value during an execution. To ensure that the execution value of a value is still well defined, we add a parameter i that signifies, which of the possible execution values we want to refer to.

Definition 4.0.1 (Sequential Execution Value)
For a program p and an input value h, we define the sequential execution value function as

$$[\![p]\!]_h : \text{VAL}_p \times \mathbb{N} \rightarrow \{0,1\}^w \cup \{\bot\}.$$

For a value v, $[\![p]\!]_h(v,i)$ is the numerical value of the i-th assignment of v during the execution. If there is no i-th assignment of the value, the function returns $\bot$.

Execution Condition $exec(\cdot)$ in Loops In our analysis, we treat loops as if they were separate functions. Therefore, when computing the execution condition $exec(\cdot)$ for the basic blocks of a program, we need special handling of the blocks inside a loop:

The execution condition for the loop header will be computed in the standard way.

For the blocks inside the loop, we assume they form a separate function with the loop header being the entry block. We compute the execution condition for those blocks in the standard way, but under the assumption that $exec(b) = \texttt{true}$ for the loop header b.

Since we treat loops as a function rather than a control flow structure, we annotate the edge between the loop head b and the block following the loop header after the loop $\hat{b}$ with $follow((b,\hat{b})) = \texttt{true}$. The execution condition $exec(\hat{b})$ is computed using this annotation.

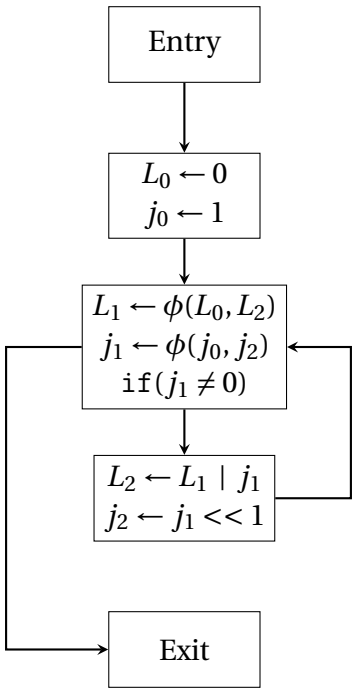
4.1. Loops

Loops are handled during the dependency analysis with the following steps:

Input $\emptyset$
Output L: int

1: $\mathrm{L} : int \leftarrow 0$
2: $j : int \leftarrow 1$
3: **while** $j \neq 0$ **do**
4: $\quad \mathrm{L} \leftarrow \mathrm{L} \mid j$
5: $\quad j \leftarrow j << 1$
6: **end while**

(a) Program code before SSA-transformation



(b) CFG of the example program in SSA-form

Figure 4.1.: Example program for Loop Analysis. The program doesn't have a secret input and always returns 111_2.

First, we isolate the loop from the rest of the program and analyze the loop body as a separate function. Second, we generate the dependency vectors for loop output values. These dependency vectors act as a map from the loop inputs to the loop outputs. We compute the outputs of specific iterations using this map. Finally, we combine the iteration results into dependency vectors that represent the effect of the loop during the execution as a whole.

Because all three steps refer to the same program values in a different context, we use different notations to differentiate between them. The notations represent the value as an element of VAL_p and are used as alternative names for the values during different stages of the analysis. For a program value v that is defined inside a loop l:

- $v[l]$ is used instead of v during the general loop body analysis, which is the analysis of the loop separate from the rest of the program.
- $v[l,i]$ refers to the value v in the i-th iteration of the loop l.
- v is the value v in its state after the execution of the loop is finished

General Loop Body Analysis We begin the analysis by defining input and output values for a single loop iteration.

The set of values that are used inside a loop can be divided into two categories:

1. Values that might change with each iteration. These are values that are defined inside the loop (or before the loop in the case of the first loop iteration) and then get passed to the next iteration via a ϕ-function in the loop header.

2. Values that are constant for every loop iteration.

The first group of values represents the inputs and outputs of a loop iteration, while the second group can be treated as constants in the loop analysis, even if their concrete value is unknown.

Definition 4.1.1 (Loop Inputs and Outputs)
Let l be a loop in program p.

(a) We define the set $in[l]$ of inputs of the loop as the set of values defined by ϕ-functions in the header of the loop l.

(b) We define the set $out[l]$ of outputs of the loop as the set of values that are defined inside the loop *and* appear as arguments in a ϕ-function in the loop header of l.

Analogous to the notation for values, $in[l]$ and $out[l]$ refer to the input and output sets during the general loop body analysis. The input and output sets for the i-th iteration are called $in[l,i]$ and $out[l,i]$.

Using the inputs and outputs, we can apply the principles of the basic dependency analysis to the loop l. We represent the values of the inputs as vectors of fresh boolean variables by setting $dVec(v[l]) = \mathcal{V}ar(v[l])$ for $v[l] \in in[l]$. Next, we compute the dependency vectors of all values defined inside the loop.

The dependency vectors of the loop outputs $out[l]$ are made up of formulas that depend on

1. Variables that represent loop input bits
2. Variables that represent program input bits
3. Variables that represent loop input bits from enclosing loops

Additional to the inputs and outputs, we define a propositional formula $exit[l]$ for the loop l, which represents the condition that must be fulfilled to jump out of the loop. Like the formulas for values in $out[l]$, $exit[l]$ depends on variables from the loop inputs, the program inputs, and enclosing loop inputs.

Example Figure 4.1 shows an example of a program containing a loop l. The inputs of the loop are the values $L_1[l]$ and $j_1[l]$. The outputs of the loop are the values $L_2[l]$ and $j_2[l]$.

The dependency vectors of those values are:

$$
\begin{aligned}
dVec(L_1[l]) &= [L_1^2, L_1^1, L_1^0] \\
dVec(j_1[l]) &= [j_1^2, j_1^1, j_1^0] \\
dVec(L_2[l]) &= [L_1^2 \vee j_1^2, L_1^1 \vee j_1^1, L_1^0 \vee j_1^0] \\
dVec(j_2[l]) &= [j_1^1, j_1^0, 0]
\end{aligned}
$$

The loop condition is $exit[l] = \big([j_1^2, j_1^1, j_1^0] == [0,0,0]\big)$.

Computing Iterations We use the dependency vectors from the general loop body analysis, to obtain dependency vectors that represent the loop outputs after a specific iteration. The formulas in these vectors will then only depend on the program input variables and not the loop variables.

Input and output values of specific loop iterations are collected in the sets $in[l,i]$ and $out[l,i]$, where i is the iteration count. The special case $i = 0$ refers to the program state before the loop is entered. We leave $in[l,0]$ undefined and use $out[l,i]$ as the values that are computed before the loop and then used inside the loop during the first iteration. For $i > 0$ we have $in[l,i+1] = out[l,i]$.

Given the dependency vectors for values in $in[l,i]$ for some value of i, we can compute the dependency vectors for values in $out[l,i]$, by substituting the variables representing values in $in[l]$ by the corresponding formula of the value in $in[l,i]$.

Definition 4.1.2 (Iteration Substitution)
Let p be a program with a loop starting at block l. We are given dependency vectors for values in $in[l,i]$. To compute the dependency vectors for values in $out[l,i]$, we use the substitution

$$\sigma_{l,i} := \{h[l]^j \mapsto dVec(h[l,i])^j \mid \quad h[l] \in in[l]\}$$

and apply it to the dependency vectors of $out[l]$.

The substitution $\sigma_{l,i}$ can be applied to $exit[l]$ to obtain a formula $exit[l,i]$ that represents the condition of whether the loop is exited after the i-th loop iteration.

Example (cont'd) The dependency vectors of the input and output values, as well as the loop condition, for the first iterations for the example program are presented in table 4.1. The construction of the substitution for $i = 1$ is shown below:

Iteration	Inputs		Outputs		Exit Condition
	$L_1[l,i]$	$j_1[l,i]$	$L_2[l,i]$	$j_2[l,i]$	$exit[l,i]$
$i=0$	-	-	[000]	[001]	[001] == [000]
$i=1$	[000]	[001]	[001]	[010]	[010] == [000]
$i=2$	[001]	[010]	[011]	[100]	[100] == [000]
$i=3$	[011]	[100]	[111]	[000]	[000] == [000]

Table 4.1.: Dependency Vectors for the values of the sets $in[l,i]$ and $out[l,i]$ for the example program 4.1b.

$$
\begin{aligned}
dVec(L_1[l]) &:= [\ L_1^2 \quad L_1^1 \quad L_1^0\ ] \\
dVec(L_1[l,0]) &:= [\ 0 \quad 0 \quad 0\ ] \\
\sigma_{l,1,L} &:= \{\ L_1^2 \mapsto 0 \quad L_1^2 \mapsto 0 \quad L_1^0 \mapsto 0\ \} \\
dVec(j_1[l]) &:= [\ j_1^2 \quad j_1^1 \quad j_1^0\ ] \\
dVec(j_1[l,0]) &:= [\ 0 \quad 0 \quad 1\ ] \\
\sigma_{l,1,i} &:= \{\ j_1^2 \mapsto 0 \quad j_1^2 \mapsto 0 \quad j_1^0 \mapsto 1\ \} \\
\sigma_{l,1} &:= \sigma_{l,1,L} \cup \sigma_{l,1,i}
\end{aligned}
$$

The dependency vectors for the values $j_2[l,1]$ and $L_2[l,1]$ are the result of applying the substitution $\sigma_{l,1}$ to the vectors $dVec(j_2[l])$ and $dVec(L_2[l])$.

Combining Iterations for Overall Loop Result To complete the loop analysis, we compute dependency vectors for the output values, that represent the loop execution as a whole.

The result values of the loop depend on the number of iterations that are executed. The case that the loop is executed exactly i times is represented by the condition $iterations[l,i]$:

Definition 4.1.3 (Loop Iterations)
Let p be a program with a loop beginning with basic block l. The propositional formula

$$iterations[l,i] : \left(\bigwedge_{0 \le j < i} \neg exit[l,i] \right) \wedge exit[l,i]$$

is a propositional formula that evaluates to true, iff for a program input h the loop is executed exactly i times. IT contains variables from $\mathcal{V}ar_p$ and variables representing inputs from enclosing loops where applicable.

The formula $iterations[l,i]$ encodes the following intuition: For every $j < i$, $exit[l,j]$ must be false, otherwise the loop execution would have been aborted before

Algorithm 4.1 Loop Result Computation

Input Iteration results $v[l,i]$ for the loop output value v and iteration conditions $iterations[l,i]$

Output $dVec(v)$: Dependency vector that represents the overall computation result of the loop for the value v.

1: $dVec(v) \leftarrow \perp$ // initialize with placeholder value
2: $i : int \leftarrow 0$
3: **while** $i < maxIter$ **do**
4: $\quad dVec(v) \leftarrow dVec(v).replace(\perp, \mathbb{IF}(iterations[l,i], dVec(v[l,i]), \perp))$
5: $\quad i++$
6: **end while**

Figure 4.2.: Algorithm for the computation of loop output values. The function `o.replace(x, y))` replaces occurrences of `x` in expression `o` with `y`

the i-th iteration. The condition $exit[l,i])$ must be true, otherwise, the loop would have been executed more than i times.

For the loop output values, we make the following observations:

We can check whether the loop is executed exactly i times or not via the condition $iterations[l,i]$. If the loop is executed exactly i times, the results are then equal to the outputs $out[l,i]$. Continuing this observation for more iterations leads to the definition of algorithm 4.1that computes the dependency vector for a loop output value. To ensure the termination of the algorithm, we have to limit the number of loop iterations it computes to an upper bound $maxIter$. The following paragraph discusses the soundness of this approximation.

Approximation: Limiting Loop Iterations Limiting the number of loop iterations means, that we exclude certain program inputs h from the analysis, namely those that require more than $maxIter$ iterations. If such an input is used to evaluate the dependency vector of a loop output value v, $dVec(v)$ will evaluate to $\perp$. We interpret $\perp$ as an invalid execution and disregard the value in the following analysis. If the dependency vector $dVec(v) = \perp$ is used in the computation of the dependency vector for another value v', $dVec(v')$ will also be set to $\perp$.

The exclusion of certain input values means, we have to adjust the equivalence from theorem 3.1:

Theorem 4.1 (Weakened Equivalence Theorem)
Given a program p, a program input value h and a program value v that is defined outside any loops and for which $[\![p]\!]_h(v) \neq \perp$. The dependency vector of the value v fulfills the following condition:

$$\mathcal{V}_h(dVec(v)) \neq \bot \implies (\forall 0 \leq i < w : \mathcal{V}_h(dVec(v)^i) \iff [\![p]\!]_h(v)^i)$$

The correctness of this theorem can be proven similarly to the proof given in appendix B. The premise $\mathcal{V}_h(dVec(v)) \neq \bot$ implies, that the execution path for the input h until the definition of the value v was completely analyzed and the input h is not part of the inputs that are excluded from the analysis by the loop iteration limit.

To compute the dynamic leakage of the execution and the channel capacity, we must adjust the method from section 3.3 to take into account the weakened equivalence theorem. Let p be a program that contains a loop. We consider an execution that produces the output value l.

Dynamic Leakage Any input value h for which $\mathcal{V}_h(dVec(\mathtt{L}))$ doesn't evaluate to $\bot$ still fulfills theorem 3.1. That means we can correctly decide if $h \in \mathscr{H}_l$. If we have an input value h' for which $\mathcal{V}_{h'}(dVec(\mathtt{L})) = \bot$, our analysis is not able to decide whether $h' \in \mathscr{H}_l$. However, we can safely approximate the set $\mathscr{H}_l$, by not adding any inputs with invalid executions to $\mathscr{H}_l$. Under-approximation is sound because the knowledge gained by the attacker increases as the size of $\mathscr{H}_l$ decreases. Therefore the approximated dynamic leakage is a safe upper bound for the information leaked by the execution.

Channel Capacity The channel capacity analysis tries to find all possible outputs l. If we restrict the set $\mathscr{H}$ from which the inputs are taken, we possibly also restrict the set of possible outputs $\mathscr{L}$. To safely approximate the size of the set $\mathscr{L}$, we must assume that every input h with $\mathcal{V}_h(dVec(\mathtt{L})) = \bot$ produces a distinct output l that is not the result of any other program execution. Over-estimating the size of $\mathscr{L}$ means over-estimating the amount of information an attacker might gain and is therefore sound.

Example (cont'd) Applying algorithm 4.1 to the value L_2 from example 4.1b, gives the following results:

$$\begin{aligned}
dVec(L_2) \quad &= \quad \mathbb{IF}(([001] == [000]), \quad [000], \\
&\qquad\quad \mathbb{IF}(([010] == [000]), \quad [001], \\
&\qquad\quad \mathbb{IF}(([100] == [000]), \quad [011], \\
&\qquad\quad \mathbb{IF}(([000] == [000]), \quad [111], \quad \bot))))) \\
&= \quad \mathbb{IF}((\mathtt{false}), \quad [000], \\
&\qquad\quad \mathbb{IF}((\mathtt{false}), \quad [001], \\
&\qquad\quad \mathbb{IF}((\mathtt{false}), \quad [011], \\
&\qquad\quad \mathbb{IF}((\mathtt{true}), \quad [111], \quad \bot))))) \\
&= \quad [111]
\end{aligned}$$

4.2. Functions

We assume that all functions that are part of the input program are pure, i.e. they have no side effects. This means the only way for information to flow into and out of the function is through the input parameters and the return value.

We write $\texttt{Func}_p$ for the set of functions that belong to a program p. For the analysis of the function f, we treat f as its own program, where the parameters correspond to the input `H` and the return value corresponds to the output `L`.

Function Analysis In the following, we first consider the analysis of non-recursive functions with a single return statement. Independent of any call sites of the function, we use the standard dependency analysis algorithm to compute dependency vectors for all values inside the function.

The dependency vectors computed for values of this function are defined over the variables created to represent the function's parameters. They do not contain variables representing bits of values from outside the function.

Dependency Analysis for `call`-Statements The statement $v \leftarrow \texttt{call}\, f(a)$ calls the function f with the arguments $a := (a_0, ..., a_m)$ and assigns the return value of the call to the value v. To compute the dependency vector $dVec(v)$, we substitute the variables in $\mathcal{V}ar_f$ with the dependency vectors of the arguments:

Definition 4.2.1 (Call Site Substitution)
Let $f \in \texttt{Func}_p$ be a function in p that has input parameters $\texttt{P} := (\texttt{P}_0, ..., \texttt{P}_m)$ and return value r_f and let the expression $\texttt{call}\, f(a)$ be a call to f with the arguments $a := (a_0, ..., a_m)$.

The substitution $\sigma_{f(a)}$, defined as

$$\sigma_{f(a)} := \{dVec(\texttt{P}_i)^j \mapsto dVec(a_i)^j \mid \texttt{P}_i \in \texttt{P}\}$$

is called the *call site substitution* of the expr $\texttt{call}\, f(a)$ and substitutes the variables in $dVec(\texttt{P}_i)$ representing the input parameters $\texttt{P}_i$ by the corresponding boolean predicates of the dependency vectors belonging to the call site's arguments (note that $dVec(P_i)$ contains only variables that represent the bits of the function parameter P_i).

The definition of $\mathscr{E} : \texttt{Expr} \rightarrow \mathscr{F}$ for function call expressions is then given as:

$$\mathscr{E}(\texttt{call}\, f(a)) := \sigma_{f(a)}(dVec(r_f))$$

Lemma 4.2
Using the extended definition of $\mathscr{E}$ including call-expressions, theorem 3.1 (theorem 4.1 in case of loop approximation) is still fulfilled.

4.3. Functions with Multiple Return Statements

A function f may have more than one return statement. Those statements in general return different values $v_1, ...v_k$. Thus it is necessary to combine the dependency vectors of these values into a single vector r_f that accurately represents the returned value based on the arguments of this function. We assume that return statements are not contained in a loop.

The combining of multiple return values requires the following observation: If $b_1, b_2 \in \mathrm{BB}_f$ are basic blocks that each contain a return statement and $b_1 \neq b_2$, then $exec(b_1) \wedge exec(b_2)$ is unsatisfiable. If this were not true, and $exec(b_1) \wedge exec(b_2)$ was satisfiable, then lemma 3.4 tells us, that there must be a set of function arguments for f, for which both b_1 and b_2 are executed. This would mean the execution of f has multiple return values, which is an obvious contradiction. Thus, $exec(b_1) \wedge exec(b_2)$ must be unsatisfiable.

It follows directly from this observation that if a function has k return statements which appear in the blocks $b_1, ..., b_k$, at most one of the conditions $exec(b_1), ..., exec(b_k)$ can be fulfilled for any truth assignment. Simultaneously, every execution of a function with a return value includes a return statement. Hence exactly of the conditions $exec(b_1), ...exec(b_k)$ must be fulfilled for any truth assignment.

Let $v_1, ...v_k$ be the values that are possibly returned by a function. The return statements are located in $b_1, .., b_k$ respectively. We compute the vector r_f in such a way that, if for a function argument a the condition $\mathcal{V}_a(exec(b_i))$ is true, $\mathcal{V}_a(r_f)$ will be equivalent to $\mathcal{V}_a(dVec(v_i))$. We define the function $select : 2^{\mathrm{VAL}_f \times BB_f} \rightarrow \mathscr{F}^w$ that fulfills this condition:

Definition 4.3.1 (Value Selection)
Let $s = \{s_0, ..., s_k\} \in 2^{\mathrm{VAL}_f \times BB_f}$ be a set of tuples where $s_i = (v_i, b_i)$. We define

$$select : 2^{\mathrm{VAL}_f \times BB_f} \rightarrow \mathscr{F}^w$$

$$select(s) = \begin{cases} dVec(v) & |s| = 1 \text{ and } s = (v, b) \\ \mathbb{IF}(exec(b_i), dVec(v_i), select(s \setminus s_i)) & |s| > 1, s_i = (v_i, b_i) \in s \text{ arbitrary} \end{cases}$$

$select(s)$ corresponds to a sequential application of $\mathbb{IF}(\cdot,\cdot,\cdot)$

Example Figure 4.3 shows a program that contains a function with three return statements. The possible return values in the example program are 0 in block b_1, 1 in block b_3 and 2 in block b_4. The dependency vectors and execution conditions for the function isEvenGreaterZero() are:

$$
\begin{array}{lcrlr}
dVec(x) & = & [x^2 x^1 x^0] & & \\
dVec(0) & = & [000] & & \\
dVec(1) & = & [001] & & \\
dVec(2) & = & [010] & & \\
 & & & & \\
exec(b_1) & = & x^2 \iff \texttt{false} & & \\
exec(b_3) & = & x^2 \not\Longleftrightarrow \texttt{false} & \wedge & x^0 \iff \texttt{false} \\
exec(b_4) & = & x^2 \not\Longleftrightarrow \texttt{false} & \wedge & x^0 \not\Longleftrightarrow \texttt{false}
\end{array}
$$

The return vector of the function isEvenGreaterZero() is given by:

$$\begin{aligned} select((0,b_1),(1,b_2),(2,b_3)) &= \mathbb{IF}(exec(b_1),0, \quad select((1,b_2),(2,b_3)) \\ &= \mathbb{IF}(exec(b_1),0, \quad \mathbb{IF}(exec(b_2),1, \quad select((2,b_3))) \\ &= \mathbb{IF}(exec(b_1),0, \quad \mathbb{IF}(exec(b_2),1, \quad 2) \end{aligned}$$

4.4. Recursion

We restrict the analysis of recursive functions in this section to functions that contain at most one recursive call. The analysis we present can be extended to functions with an arbitrary number of recursive calls.

To analyze recursive functions, we begin by applying the standard function analysis described in 4.2 and 4.3. However, call statements that recursively invoke the function that they are part of, are not handled in the standard way. Instead, we use placeholder variables $r_{rec} := [r_{rec}^{w-1}, r_{rec}^{w-2}, ..., r_{rec}^{0}]$ as the return value of the call.

Simulating Recursive Calls The analysis of a recursive function f yields a dependency vector for the return value r_f of f, which depends on the variables representing the input bits and contains placeholder variables from the vector r_{rec}. These variables represent the return value r_f of the recursive call inside the function.

Under our assumption that all program executions terminate, the vector $dVec(r_f)$ will not contain variables from r_{rec} for at least one function argument. If this wasn't the case, then there would be no way to end the recursion during the execution.

For notation, we use $r_f[i]$ to mean the value r_f after the function f was executed with a maximum recursion depth i. The return value we have computed so far is $r_f[0]$. We analyse recursive calls up until a recursion bound $recBound$ and write r_f for $r_f[recBound]$.

To simulate the information flow of the recursive function call in the function f, we replace the placeholder value r_{rec} with $\sigma_{f(a)}(dVec(r_f))$, where $\sigma_{f(a)}$ is the call site substitution defined in 4.2.1 for the recursive call of f with arguments a. The result vector of the replacement operation represents the execution of function f including one recursive call and the placeholder value r_{rec} for any further recursive calls.

To simulate more than one recursive call, we repeat the same replacement operation until we reach a predefined recursion bound $recBound$. The algorithm to compute the final dependency vector of the return value of a recursive function is shown in figure 4.4.

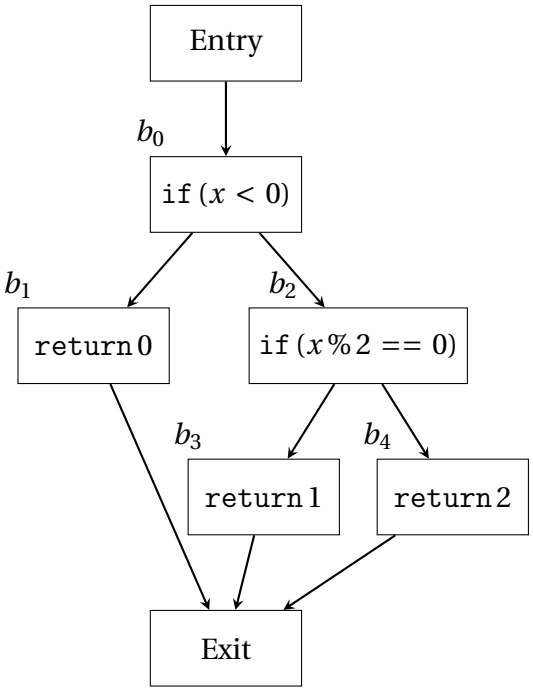
Approximation: Limiting Recursion Depth By limiting the recursion depth to $recBound$, we may exclude inputs $h \in \mathcal{H}$ from the analysis, if the execution of p with input h requires more than $recBound$ recursive calls. The effects of this approximation

```
Input H : int
Output L : int

1: L ← isEvenGreaterZero(H)

2: procedure isEvenGreaterZero(x : int): int
3:     if x < 0 then
4:         return 0
5:     else
6:         if x % 2 == 0 then
7:             return 1
8:         else
9:             return 2
10:        end if
11:    end if
12: end procedure
```

(a) Source code of the example



(b) CFG for the function isEvenGreaterZero(·)

Figure 4.3.: Example program containing a function with multiple return statements

Algorithm 4.2 Recursive Function Return Value Computation

Input $dVec(r_f[0])$: Dependency vector of f's return value,
containing r_{rec} as a placeholder.
$\sigma_f(a)$:Call site substitution for recursive call `call` $f(a)$
Output $dVec(r_f)$: Dependency vector of f's return value.

1: $i : int \leftarrow 0$
2: **while** $i < recBound$ **do**
3: $\quad dVec(r_f[i+1]) \leftarrow dVec(r_f[i]).replace(r_{rec}, \sigma_{f(a)}(dVec(r_f[0])))$
4: $\quad i++$
5: **end while**

Figure 4.4.: Algorithm for the computation of return values of recursive functions. The function `o.replace(x, y))` replaces the elements of the vector `x` in expression `o` with the elements of the vector `y` that have the same index.

on the computation of the information leakage of p are the same as for the approximation in the loop analysis described in 4.1. We treat the approximations of the dynamic leakage and the channel capacity in the same manner as in the approximation for loop iterations.

4.5. Break-Statements

To analyze loops that contain break statements, the loop analysis from section 4.1 has to be adapted at two points:

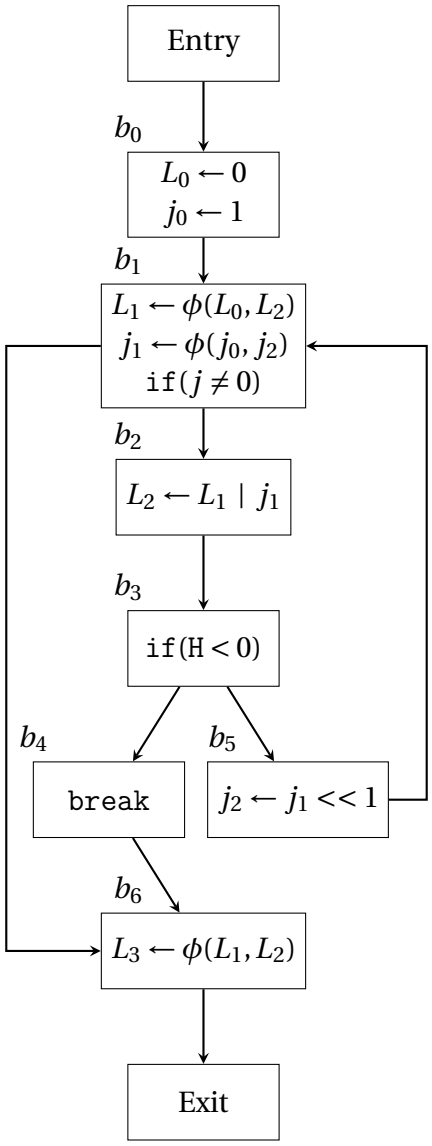
1. Which values act as the output values of the loop now depends on the point at which the loop is exited: At the end of the loop body or a break point?
2. When the loop is exited is now also determined by whether or not a break statement is executed.

Loop Output Values The identification of the loop output values happens during the general loop body analysis, where we analyze the loop body independent of the rest of the program. The goal is to identify values, that represent the output of a single loop iteration. We continue to use the notations from section 4.1.

Previously we identified the output values of a loop by analyzing the arguments of the ϕ-functions inside the loop header. For every input value, an output value is identified. The input value is mapped to this output value. The mapping represents the execution of the loop. When the loop contains a break statement, the output value depends

```
   Input H : int
   Output L: int
1: L : int ← 0
2: j : int ← 1
3: while j ≠ 0 do
4:     L ← L | j
5:     if H < 0 then
6:         break
7:     end if
8:     j ← j << 1
9: end while
```

(a) Program code before SSA-transformation



(b) CFG of the example program

Figure 4.5.: Example program for Loop Analysis. The program returns 001_2 if $H < 0$, otherwise 111_2.

on whether the execution of the loop body is completed or exited early via a break statement.

Let $v[l]$ be an input value to a loop. If the loop is without break statements we can map $v[l]$ to a single loop output value $v'[l]$. If the loop does contain break statements, we first collect the set of possible output values for $v[l]$: For each location, the loop could be exited at, the set contains the value that $v'[l]$ should be mapped to if the exit is taken. This value can be found by analyzing whether the block following the loop exit contains a ϕ-function for this value and if yes, which arguments the ϕ-function has.

If no ϕ-function is present, the output value at this point is the same as the input value of the loop. If a ϕ-function is present, the output value at this point is the argument that belongs to the control flow of the basic block containing the break statement.

After having collected all possible output values for a loop input value, we combine the dependency vectors of those values into a single propositional vector that represents the overall output value for the aforementioned input value. The method for combining the dependency vectors is the same as for the combination of return values during the function analysis. When applying the $select(\cdot)$ function, we arrange the values in such a way, that this output value is chosen iff all execution conditions of the break blocks evaluate to false.

Example An example program containing a `break`-statement is given in figure 4.5. The loop in the example program has the input values L_1 and j_1. For each of those, there are two possible output values: one value for exiting the loop via the break statement and one value for exiting the loop normally. For the value L_1 both of those output values are equal. L_1 is mapped to the output value L_2. The input value j_1 is mapped to itself, if the loop is exited via the `break` and to j_2 otherwise.

The execution condition $exec(b_4)$ of the block containing the break is given by $exec(b_4) := \mathtt{H} < 0$. Not that the execution conditions for blocks inside a loop are computed separately from the rest of the program.

The propositional vectors for the output values of the loop are finally given by:

$$\begin{array}{rlll} \text{Output value for input } L_1\text{:} & \mathbb{IF}(\mathtt{H} < 0, L_2, L_2) & = & L_2 \\ \text{Output value for input } j_1\text{:} & \mathbb{IF}(\mathtt{H} < 0, j_1, j_2) & & \end{array}$$

Exiting the Loop A loop that contains break statements is exited if the loop condition in the header is unfulfilled or if a break statement is executed. A break statement is executed if the condition $exec(b)$ for the block b that contains the break statement is fulfilled. If $b_0, ..., b_k$ are all blocks in the loop that contain a break statement, the loop is exited if $exit[l] \vee \left(\bigvee_{0 \leq i \leq k} exec(b_i) \right)$ is fulfilled. This extended exit condition replaces the simple exit condition $exit[l]$ during the loop analysis.

4.6. Arrays

The analysis can support fixed-size arrays with value semantics. For fixed-size arrays, the length must be known at compile time.

Before the dependency analysis begins, arrays are turned into a set of values that each represent an array element. The values have to be transformed into SSA-form. New array value copies are introduced at the instantiation of the array as well at every `write` instruction. Even though a `write` only modifies a single array element, we generate new value copies for all array elements. This is necessary because in general, it is not possible to know which array element is modified.

For notation, we use superscript indices $(a^0, a^1, ...)$ for the values representing the elements of the array a. Subscript indices $(a_0^0, a_1^0, ...)$ indicate the copies of those variables created during the SSA transformation.

The new values represent array entries for which we compute dependency vectors that describe the execution value of the array entry, analogous to the other values.

Array Instantiation If a new array is instantiated, all values are initialized with `0`. Their dependency vectors are constant vectors containing the value `false`.

Array Write Consider the expression $a[k] \leftarrow e$, where we write the result of the expression e to the k-th element of the array a. Let a be an array of length m. The values for the array entries before the write instruction are called $a_i^0, a_i^1, ..., a_i^{m-1}$ and the values for the array entries after the write instruction are called $a_{i+1}^0, a_{i+1}^1, ..., a_{i+1}^{m-1}$.

The dependency vectors of the values $a_{i+1}^0, a_{i+1}^1, ..., a_{i+1}^{m-1}$ are given by:

$$dVec(a_{i+1}^j) := \mathbb{IF}(j == k, \mathscr{E}(e), a_i^j), \quad i = 0...m-1$$

The expression assigns the array entry value the propositional vector $\mathscr{E}(e)$ if the index j of the entry corresponds to the write-index k. Otherwise, the array entry is not changed.

Array Read Consider the expression $v \leftarrow a[i]$, where we assign the value of the i-th element of the array a to the value v. Let $a := [a^0, ..., a^m]$ be the array entry values that represent the array a at the read instruction.

We define a function $choose(\cdot,\cdot) : \mathbb{N} \times 2^{\text{VAL}_p} \rightarrow \text{VAL}_p$ that accepts an integer i and an ordered set S of values as inputs and returns the i-th element of S. Using this function we define the dependency vector for the value v as:

$$dVec(v) := choose(i, a) \text{ a}$$

5. Hybrid Analysis

Using the methods from the previous chapters, we are able to measure a program's channel capacity, as well as the leakage of a single program run.

In this section, we will introduce techniques to integrate static analyses (Nildumu [15] and JOANA [31]) into the algorithm in order to decrease the computational load that is necessary to obtain the final leakage.

5.1. Static Pre-Processing

To keep the effort of computing the dependency formulas, as well as their evaluation with the model counter, as low as possible, we statically pre-process the input program to identify those statements that do not need to be included in the dependency analysis.

The pre-processing consists of the three stages shown in figure 5.1. In the following section, we will use the program from figure 5.2 as a running example to demonstrate the effects of the pre-processing.

Constant Bit Analysis We use *Nildumu* [15] to perform a constant bit analysis on the input program. The goal is to identify values that are *effectively constant*. Effectively constant values have the same execution value in every run of p, regardless of the inputs that were used.

Definition 5.1.1 (Effectively Constant Value)
A program value v of the program p is called *effectively constant* iff:

$$\forall h_1, h_2 \in \mathcal{H} : [\![p]\!]_{h_1}(v) = [\![p]\!]_{h_2}(v)$$

If a value is effectively constant, we can safely exclude it from any further analysis and set its dependency vector to a vector of boolean constants that corresponds to its execution value. For values that are not effectively constant, but contain constant bits, we can also reduce the number of dependency formulas we need to compute to those bits that are not constant.

PDG Pruning If a value is effectively constant, an observer cannot learn anything about the secret inputs of a program by observing the behavior of that particular value. Since we are only interested in information flow that will help an attacker in learning our secret, the data dependencies of effectively constant values can safely be ignored. We prune the PDG of the analyzed program by removing all incoming data edges of nodes that define effectively constant values. Control dependency edges that start at a loop

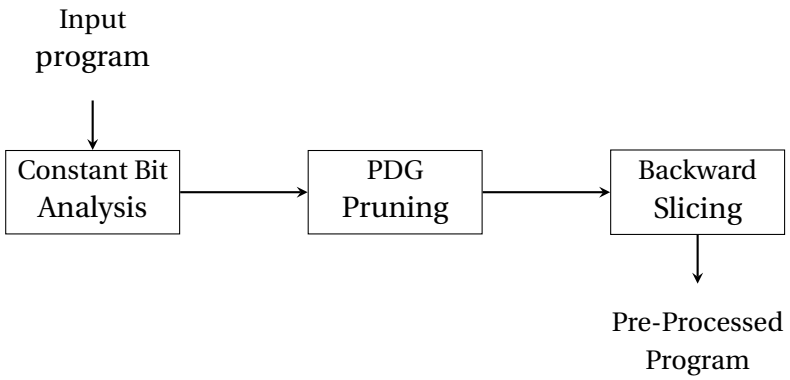


Figure 5.1.: Stages of the pre-processing pipeline. The pre-processed program is the input for the following dependency analysis.

Input H : int
Output L: int

1: **if** H < 0 **then**
2: L ← $f(-\mathtt{H}, 1)$
3: **else**
4: L ← $f(\mathtt{H}, 1)$
5: **end if**

Figure 5.2.: Assume that $f(\mathtt{int}\, x, \mathtt{int}\, y) : \mathtt{int}$ is a function, that returns the second parameter y. The value L in this program is *effectively constant*, since its execution value will always be 1 every input H.

header (an if-statement header) can be removed if all values that result from the loop (from the two branches) are effectively constant. This condition is easily verifiable in JOANA, as JOANA adds an additional edge from the header of the control flow structure to the block where the paths merge. Figure 5.3 shows the original and the pruned version of the PDG of the program in figure 5.2.

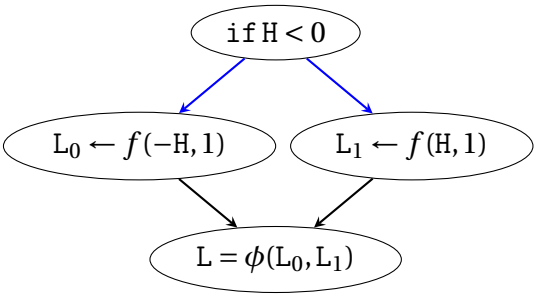
Backward Slicing As the last step, we calculate a backward slice with the slicing criterion $\langle s, v \rangle$ being the value v that is leaked to a public channel combined with the statement s of the leak. For slicing, we use the pruned PDG from the previous stage. In our analysis, we used a static inter-procedural backward slicing algorithm from the JOANA framework. The resulting backward slice contains those statements, that are needed for computing the dependency vector of the leaked value. Program statements that are not part of the slice do not have to be analyzed. Control structures, such as loops or conditional statements can be omitted if the head of the structure is not contained in the backward slice: If the head is not part of the backward slice, the resulting output value does not depend on the truth value of the expression. Therefore it also doesn't depend on any computations that are contained in the control structure. In this case, we will also omit them from the computation of the path conditions that keep track of implicit information flows.

Omitting certain statements from the dependency analysis is safe, as long as we can guarantee, that we have enough information to determine the dependency vectors of the values defined in the remaining statements. Enough information, in this case, means that the dependency vectors of all used values of the expression defining the value are known. Each use-value falls into one of the following categories:

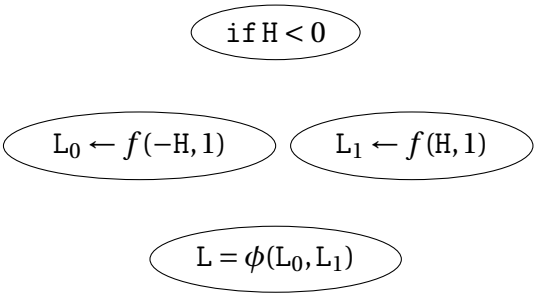
1. *Constants:* The dependency vector is constant and corresponds to the constants twos-complement representation.
2. *Parameters:* Parameters are unknown values whose dependency vectors are filled with placeholder variables.
3. *Effectively Constant Values:* The dependency vector is constant and corresponds to the twos-complement representation of the value determined by the constant bit analysis.
4. *Variable Values:* Since an expression containing the value is part of the backward slice, the definition of this value will also be included. Thus, we will have computed the value's dependency vector prior to analyzing the current expression.

Therefore it is indeed safe to omit statements in our analysis that were not included in the final backward slice of the pre-processing.

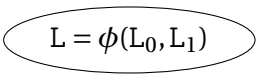
By using this pre-processing method we can shrink the propositional formulas that are produced by the dependency analysis. In the example program, the dependency vector of the value L is reduced to $dVec(\mathtt{L}) = [001]$, whereas without the pre-processing it would include a ternary operator: $dVec(\mathtt{L}) = \mathbb{IF}(\mathtt{H} < 0, [001], [001])$. This helps to increase efficiency in two ways: Firstly, the formulas the program needs to handle



(a) Before Pre-processing. The blue edges are control dependencies, while the black edges are data dependencies.



(b) After Pruning. The data edges were removed, because the value L is effectively constant. The control dependencies were removed, because the value L, which is the value resulting from the conditional after the two branches merged, is effectively final.



(c) After Slicing. The backward slice for the criterion $\langle \mathtt{L} = \phi(\mathtt{L}_0, \mathtt{L}_1), l \rangle$ now only contains a single node and is the result of the pre-processing.

Figure 5.3.: The PDG of the program in figure 5.2, at different stages during the pre-processing.

become smaller and thus take less time to process, and secondly, the computation time of ApproxMC decreases with the decrease of the size of the input formula. A more in-depth analysis of the effects of the pre-processing on the cost of the analysis as a whole is given in 7.

5.2. Hybrid Analysis for Channel Capacity

The channel capacity measures the number of distinct program outputs. Measuring the channel capacity exactly is, however, not always feasible. For example, if the program contains a loop, there might be too many possible program paths for the analysis to consider.

In this case, the analysis will disregard certain program paths estimate the channel capacity based on the program paths it did consider.

With the goal of increasing the efficiency of the analysis in terms of computation time, while maintaining or even increasing the precision of the channel capacity estimation, we combine the static analysis of the tool Nildumu [15] with our dependency analysis.

The basic idea of this hybrid analysis is, to divide the program into segments. Each segment's channel capacity will be determined separately. For this purpose, we identify those segments that are infeasible for a precise and efficient dependency analysis. These segments will instead be analyzed by Nildumu. After every segment's channel capacity is analyzed, we combine the results for an overall estimation that applies to the whole program.

Program Segmentation and Segment Analysis The program is divided according to the control flow structures it contains. The structures that are isolated are:

- loops
- conditional statements together with their branches
- functions
- linear program segments.

For each segment, we compute its channel capacity. We begin by identifying the input and output values of each segment. Input values are values that are used in at least one expression but were not defined inside the segment. Output values are values that are defined inside the segment and are used in locations outside the segment.

For each segment, the computation of the channel capacity is handled in one of the following three ways:

1. The segment is analyzed using the dependency analysis from chapter 3.
2. The segment is analyzed using Nildumu.
3. The segment is recursively analyzed using the hybrid analysis.

In general, using the dependency analysis yields the most precise results, however, it is also the most costly approach in terms of computation time. The decision which analysis approach should be taken for a given program segment depends on the following factors:

If the size of the propositional formulas becomes too large, they cannot be handled in a timely fashion by the model counter. Large formulas are mainly the result of loops, where many iterations have to be taken into account, or nested and/or recursive function calls. Very long linear programs might become problematic even without loops and function calls.

The second factor is the number of segments we divided the program into. Each segment will incur a certain amount of overhead time needed to prepare the segment for the analysis and invoke the tools used in the analysis (Nildumu, ApproxMC, further dependencies).

Consolidation of Segment Results Assume we are given a program containing a loop. We split the program into 3 segments: The part before the loop p_b, the loop itself p_l and the part after the loop p_a. We compute the channel capacities for all three parts separately: k_b, k_l, k_a.

In each segment analysis, we over-approximate the possible inputs for the program section. Thus, the computed channel capacities are possibly larger than they actually are. Because the attacker knowledge increases as the channel capacity increases, this is a sound upper limit for the information leakage of the program segment.

Because the programs we examine are deterministic, the number of outputs is always less or equal to the number of inputs of a program. Furthermore, the number of outputs of a segment is equal to the number of possible inputs of the following segment. Thus, we can obtain an overall estimation of the channel capacity for the program as a whole by taking the minimum of k_b, k_l, k_a.

The approach can be generalized to an arbitrary number of segments.

5.3. Hybrid Analysis for Dynamic Leakage

In 5.2, we presented an analysis that can combine the two different approaches of our tool and Nildumu to compute the channel capacity. Naturally, the question arises whether this approach can be used for computing the dynamic leakage as well. The main idea of the approach is the segmentation of the program, which reduces the size of the programs that have to be analyzed.

We have found it is not possible to efficiently compute the dynamic leakage with the approach of dividing the program for the following reason:

While for the channel capacity it was enough to know how many different values could potentially be transmitted between two adjacent segments, for the dynamic leakage it is essential to know which values might be transmitted. Thus, we cannot separate the segments from one another by treating the transmitted values as fresh inputs.

6. Implementation

For the implementation, we integrated the analysis into an interpreter and built our tool on top of JOANA [40]. JOANA is an IFC tool for Java programs that includes program analysis techniques that work on PDGs and SDGs. We used Java 8 for the implementation of our tool.

Input Programs The tool deals with input programs written in Java syntax. The programs can include the language features specified in sections 3 and 4.

The user can mark a function as the entry point of the analysis by calling it from the main function of the Java file. The parameters of the entry point function are taken as the secret inputs.

Variables can be leaked to a public output channel by using the special function `Out.print(·)`. The leak is assumed to be located outside any control flow structures.

Interpreter Integration One of the main ideas of our analysis is to compute the dynamic leakage of a single program execution.

We integrated our analysis into an interpreter that executes the input program for an input supplied by the user and simultaneously computes the channel capacity for the input program and the dynamic leakage for the execution.

Analysis Pipeline The analysis is executed in a pipeline with different stages:

- The *Build* stage: The input program is first compiled using `javac` and then transformed into the necessary program representations for the analysis.
- The pre-processing stage (*PP*): See section 5.1.
- The dependency analysis stage (*DA*): In this stage, the dependency vectors of the program value are generated.
- The hybrid analysis stage (*HA*): This part of the analysis computes the channel capacity of the program using the hybrid approach outlined in section 5.2. The parameters for when the tool switches from a hybrid to a static analysis can be chosen by the user.
- The execution stage (*exec*): The interpreter executes the input program using the user-supplied arguments. This stage includes the computation of the dynamic leakage at the point where a value is leaked during the execution.

Our tool allows the user to choose whether to use the static pre-processing and the hybrid analysis or not.

Tools Our analysis and interpreter uses three tools:

JOANA is used for the generation of the needed program representations (CFGs + PDGs) as well as for the slicing operations during the pre-processing stage. The static analysis used during the pre-processing and the hybrid analysis is done by Nildumu [15]. We used the stand-alone implementation. For the model counting, we use ApproxMC [13].

7. Evaluation

We evaluate the implementation of our analysis in two ways:

We compare the analysis using the hybrid method to the analysis using only the dependency analysis to examine whether it is possible to achieve a decrease in computation time and to determine the loss in precision of the results that might occur.

Secondly, we compare our tool to two other comparable tools: Nildumu [15], because it is integrated into our tool as the static part of the hybrid analysis, and ApproxFlow [12] because it uses a SAT-based analysis that is similar to our dependency analysis, however, ApproxFlow doesn't combine this with any other tools. We were not able to obtain a working instance of Flowcheck [16] to include in the comparison.

7.1. Benchmarks

In this evaluation, we use synthetic benchmarks taken from [22], [15], [41] and [42] as well as some benchmarks we created ourselves. The benchmark programs are described in appendix C. To run the benchmarks with all four tools, we translated them into Java, C, and Nildumu's own language specification. As our tool and Nildumu don't support unsigned integers, we modified benchmarks where necessary with a condition $h \geq 0$ (the modification was applied to the benchmarks for all tools).

7.2. Tools

QIFC Interpreter (QIFCI) We evaluated the benchmarks using three different configurations:

- *QIFCI:* The dependency analysis is not combined with any static analysis.
- *QIFCI-PP:* The dependency analysis is combined with the static pre-processing
- *QIFCI-H:* The dependency analysis is combined with the static pre-processing and the hybrid analysis. The hybrid analysis is applied to all loops and function calls in the program.

Nildumu We use the stand-alone implementation of Nildumu (version `2caee6b`) for the evaluation. Apart from the recursion bound, we adopted the standard configurations provided by Nildumu.

ApproxFlow For our benchmark, we used an updated version of ApproxFlow[1] that includes a newer version of ApproxMC.

7.3. Evaluation Setup

Environment The evaluation was performed on a machine with a 4-core Intel Core i7-7500U CPU and 24GiB of RAM, running a Manjaro Linux operating system.

Benchmarking For each tool, we ran each benchmark a total of 10 times and report the average run time over all 10 executions. To minimize potential interferences of other processes running in the background, we used the command line tool `chrt` to switch to a FIFO scheduling policy and set the executions' priority to the maximum value. We use `perf` to analyze the performances of the executions.

For measuring the duration of the pipeline stages in QIFCI, we used timestamps from the tool's log files. Thus the overall run time might deviate from the run times measured with `perf`.

For each tool, we used an integer width of 32 bits. For Nildumu, ApproxFlow, and our own tool, we ran each benchmark twice: Once with a recursion and loop bound of 8 and once with a recursion and loop bound of 32.

For the tool comparison, we turned off the dynamic leakage computation of QIFCI, since it is the only tool that computes this value. Thereby all tools only analyze the channel capacity of the given input program.

7.4. Findings

Channel Capacity

The results of the benchmark runs are shown in tables 7.1 and 7.2. They show the channel capacity that was computed by the tools as well as the average run time.

ApproxFlow ApproxFlow was able to precisely compute the channel capacity for all simple benchmarks. However, the tool reports channel capacities that are too low in most of the other benchmarks.

Wrong results occur if the loop bound is lower than the needed amount of loop iterations. An example of this phenomenon is the *Laundering Attack* benchmark with a loop bound of 32. The 32 iterations will each give a different output, leading to the channel capacity estimation of $log_2 32 = 5$ instead of the actual channel capacity of 32.

In cases where the channel capacity cannot be accurately computed, the tool does not approximate safely.

[1] The ApproxFlow version we used, can be found at `https://github.com/parttimenerd/approxflow`

The run time of ApproxFlow was below 0.3s for all benchmarks. The run times were only minimally affected by the loop bound increase from 8 to 32. The complexity of the benchmarks had a greater influence on the run time. The *Sum Query* benchmark had the longest run time. We suspect this is due to the model counting process in the analysis, as we saw the same effect in QIFCI.

The low accuracy of ApproxFlow in our analysis can be attributed to the loop bound being too low. Increasing the bound will improve the leakage estimate. The run times suggest that even with a significantly higher bound, the execution would remain in an acceptable time frame.

Nildumu All channel capacities computed by Nildumu are equal or greater than the exact channel capacity, with the exception of the *Masked Copy* benchmark, where Nildumu underestimated the leakage by one bit[2]. For all other benchmarks, the approximation by the tool is safe.

Inaccurate results occur for some benchmarks containing implicit flows: In the *Implicit Flow* benchmark, the tool does not recognize that two of the branches output the same value. The benchmarks *Sanity Check* and *Sane Laundering* show the same issues.

The run times of Nildumu were significantly higher than those of ApproxFlow. Increasing Nildumu's inlining bound does not affect the run time significantly.

QIFCI QIFCI in the configuration without static pre-processing and without hybrid analysis is able to accurately compute the channel capacity for the simple benchmarks. Programs containing loops and functions are also accurately analyzed if the loop and recursion bounds are high enough to include all possible executions. If this is not the case (e.g. in *Sane Laundering* and *Parity*), the results may be inaccurate. In this case, the actual channel capacity is safely over-approximated.

The run time of QIFCI is highly vulnerable to increases in the analysis bounds used, especially for benchmarks including loops. For the *Sane Laundering* benchmark the run time almost triples.

QIFCI-PP Adding the static pre-processing to the configuration of QIFCI does not change the channel capacities that are computed.

Run times for the simple benchmarks were within a small range compared to the QIFCI configuration with the exception of the *Table Lookup* benchmark, where the run time increased significantly. Run times for loop benchmarks and function/recursion benchmarks increased most of the time. The static bit analysis in the pre-processing cannot assign many constant values to variables inside loops. This decreases the performance gain during the later dependency analysis, while the overhead incurred by the pre-processing does not change significantly.

The only decrease in run time in the loop benchmarks can be found in *Shift and Launder*. This benchmark deliberately includes computations inside a loop that are irrelevant to the output. Here the PDG Pruning based on the output's backward slice

[2] In revision `49ebe88` of Nildumu the issue was fixed and Nildumu correctly reports a leak of 16 bits.

leads to a performance gain that outweighs the additional run time required by the pre-processing.

QIFCI-H The configuration of our tool including the pre-processing and the hybrid analysis was able to produce almost identical results as the other two configurations. The only change occurs in *Sane Laundering*, where the analysis of QIFCI-H accurately returns a channel capacity of 4 for an iteration bound of 8. The other two QIFCI configurations recognized that there are program executions that do not adhere to the iteration bound of 8 and thus assumed that such executions could return arbitrary values. During the hybrid analysis, the sanity check part and the laundering part of the *Sane Laundering* are analyzed separately. The final channel capacity is 4, the channel capacity of the sanity check part.

Run times for the simple benchmarks show no significant changes compared to the QIFCI-PP evaluation. Run times for the loop benchmarks were reduced significantly compared to QIFCI-PP run times, especially with a higher iteration bound. Since in the hybrid analysis loops and recursive functions are only analyzed statically, the run time cost of adding more loop iterations / more recursion depth is almost completely eliminated. With the higher iteration bound, the QIFCI-H also executes faster than QIFCI for all loop benchmarks and one of the recursion benchmarks. Increases in run time from QIFCI to QIFCI-H are not significant compared to run time increases from increased analysis bounds or more complex benchmarks.

Hybrid Analysis

Table 7.3 shows the changes in run time between QIFCI and QIFCI-H divided into the tools' different analysis stages. The stages are described in section 6.

Build Stage The *Build* stage is not affected by the hybrid analysis. As expected the run times for both configurations are within a normal range of variance to each other.

Pre-Processing Stage and Dependency Analysis stage The evaluation of the pre-processing stage shows that the loop benchmarks, as well as the *Table Lookup* benchmark, need the most time. This corresponds with Nildumu having the longest run time for these benchmarks. The only benchmark where this correlation doesn't hold is the *Table Lookup* benchmark.

The run time needed for the *Dependency Analysis* stage was decreased by the pre-processing in every case, however, for most benchmarks, the decrease is not significant and most likely due to normal variances in run time.

Hybrid Analysis Stage During the *Hybrid Analysis* stage simple benchmarks, that contain no control flow structures that are handled by the hybrid analysis, are not significantly affected in their run time by enabling the hybrid analysis. The computation of the channel capacity is decreased significantly for the benchmarks *Laundering*, *Shift*

		ApproxFlow		Nildumu	
	CC	8	32	8	32
Masked Copy		0.16	0.15	5.23	5.20
	16	**16**	**16**	**15**	**15**
Sum Query		0.30	0.30	5.24	5.40
	32	**32**	**32**	**32**	**32**
Sanity Check		0.11	0.12	5.29	5.27
	4	**4**	**4**	**32**	**32**
Table Lookup		0.13	0.12	5.94	5.92
	3	**3**	**3**	**3**	**3**
Implicit Flow		0.12	0.11	5.46	5.33
	2.80	**2.80**	**2.80**	**3**	**3**
Parity		*error*[3]	*error*[3]	6.72	6.63
	1			**1**	**1**
Laundering Attack		0.11	0.13	6.50	6.40
	32	**3**	**5**	**32**	**32**
Shift and Launder		0.11	0.19	7.66	7.01
	32	**3**	**5**	**32**	**32**
Masked Laundering		0.14	0.17	7.57	8.49
	31[4]	**2.31**	**4.08**	**32**	**32**
Sane Laundering		0.16	0.20	7.82	7.61
	4	**2.32**	**4**	**32**	**32**
Recursive Laundering		0.12	0.23	5.62	5.61
	32	**3.16**	**5.04**	**32**	**32**
Call Mask		0.24	0.24	5.54	5.41
	28	**28**	**28**	**28**	**28**
Dead Recursion[5]		0.12	0.14	6.60	6.60
	0	**0**	**0**	**0**	**0**

Table 7.1.: Results for ApproxFlow, Nildumu, and Flowcheck. The column *CC* shows the channel capacity. For each benchmark, we show the calculated channel capacity and the average run time.

[3] ApproxFlow crashed when trying to execute the benchmark with the given loop unwinding bounds. Increasing the bound to > 32 fixed the error.

[4] The exact value of the channel capacity is $\log_2(2^{31}+1) \approx 31$.

[5] ApproxFlow crashes after correctly recognizing that the leaked value is constant.

		QIFCI		QIFCI-PP		QIFCI-H	
	CC	8	32	8	32	8	32
Masked Copy		5.26	4.53	4.92	4.84	5.06	4.91
	16	**16**	**16**	**16**	**16**	**16**	**16**
Sum Query		7.05	6.35	6.82	6.58	6.60	6.62
	32	**32**	**32**	**32**	**32**	**32**	**32**
Sanity Check		6.04	5.25	5.29	5.09	5.65	5.72
	4	**4**	**4**	**4**	**4**	**4**	**4**
Table Lookup		4.32	4.36	12.16	12.42	13.88	13.60
	3	**3**	**3**	**3**	**3**	**3**	**3**
Implicit Flow		4.61	5.55	5.17	5.86	4.96	5.82
	2.8	**2.8**	**2.8**	**2.8**	**2.8**	**2.8**	**2.8**
Parity		4.81	4.64	5.68	5.59	6.87	5.38
	1	**1**	**1**	**1**	**1**	**1**	**1**
Laundering Attack		7.74	14.24	9.33	17.56	8.50	8.60
	32	**32**	**32**	**32**	**32**	**32**	**32**
Shift and Launder		7.44	19.36	9.25	17.86	9.44	8.79
	32	**32**	**32**	**32**	**32**	**32**	**32**
Masked Laundering		6.70	10.95	8.34	12.67	8.71	8.52
	31[6]	**32**	**32**	**32**	**32**	**32**	**32**
Sane Laundering		12.48	35.85	14.53	37.79	9.65	10.30
	4	**32**	**4**	**32**	**4**	**4**	**4**
Recursive Laundering		6.01	10.36	7.76	12.08	7.88	7.80
	32	**32**	**32**	**32**	**32**	**32**	**32**
Call Mask		4.69	4.65	5.24	5.28	5.70	5.63
	28	**28**	**28**	**28**	**28**	**28**	**28**
Dead Recursion		4.74	4.66	5.39	5.09	5.89	5,63
	0	**0**	**0**	**0**	**0**	**0**	**0**

Table 7.2.: Results for the 3 benchmarked configurations of QIFCI. The analysis doesn't include the execution of the program and the computation of the dynamic leakage.

[6] The exact value of the channel capacity is $\log_2(2^{31}+1) \approx 31$.

and Launder, *Masked Laundering* and *Sane Laundering*. In part this is due to the reduced run time of ApproxMC, the biggest decrease, however, is seen in the rest of the *Hybrid Analysis* stage, where the formula for the model counter is generated from the dependency vectors and turned into CNF.

Execution Stage The *Execution* stage for simple benchmarks is not greatly affected by thy hybrid analysis. The loop benchmarks that benefited from the hybrid analysis during the *Hybrid Analysis* stage show a drastic increase in run time during the *Execution stage*. The analysis of the channel capacity and the dynamic leakage is based on the same propositional formulas. The effort that was saved in earlier stages (loop unrolling, recursive function inlining) has to be put into the formula generation for the dynamic leakage computation. The benchmark *Masked Laundering* and *Sane Laundering* have similar total run times for both configurations. The distribution of the run time is shifted towards the *Execution* stage if the hybrid analysis is employed.

The benchmark *Shift and Launder* is the only benchmark with a significant overall decrease in run time. This is due to this benchmark being the only one that contains computations that don't influence the output, thereby being the only benchmark that considerably benefits from the PDG pruning in the pre-analysis.

Dynamic Leakage

Table 7.4 shows the dynamic leakage that we computed for the benchmarks with certain selected inputs. We chose inputs that represent different equivalence classes regarding the indistinguishability relation.

We used the QIFCI configuration to compute the dynamic leakages. The computed leakages do not change with the other configurations. We ran each benchmark 4 times, with changing quantities for integer width (8, 32) and loop iteration/recursion bound (8, 32).

For many benchmarked executions, the computed dynamic leakage by QIFCI is equal to the true dynamic leakage of the given program and input. Cases, where the two values differ, can be divided into two categories:

1. *Differences due to approximated model counting*:
   The model counter ApproxMC is not able to compute a precise model count for every benchmark execution. This case occurs for cases where the number of models is close to, but not equal to, the number of possible inputs. The approximation by ApproxMC leads to an inaccurate dynamic leakage. Since ApproxMC does not guarantee whether the result is an over- or an under-approximation, this could lead to unsafe under-approximations of the leaked information. However, the error in the computed dynamic leakage for these benchmarks is small enough to not pose a safety risk.

2. *Differences due to approximated loop and recursion formulas*:
   The propositional formulas generated during the analysis include restrictions for

	Build	**PP**	**DA**	**HA**	(MC)	**Exec**	(MC)	**Total**
Masked Copy	4.13	-	0.01	0.06	0.07	0.05	0.05	4.25
	3.85	0.31	0.01	0.07	0.05	0.05	0.05	4.30
Sum Query	4.82	-	0.04	2.03	0.58	0.01	*time out*[7]	*time out*
	4.99	0.34	0.02	2.20	0.75	0.01	*time out*[7]	*time out*
Sanity Check	5.21	-	0.02	0.90	0.05	0.46	0.01	6.61
	5.02	0.41	0.02	1.11	0.07	0.36	0.02	6.94
Table Lookup	4.70	-	0.08	0.83	0.16	0.14	0.03	5.75
	4.22	8.52	0.04	0.56	0.12	0.09	0.02	13.44
Implicit Flow	4.17	-	0.07	0.11	0.02	0.02	0.01	4.38
	4.09	0.39	0.00	0.09	0.01	0.03	0.01	4.61
Parity	4.54	-	0.54	0.15	0.01	0.40	0.32	5.65
	4.11	0.83	0.38	0.78	0.24	0.34	0.23	6.47
Laundering	4.01	-	0.13	9.19	3.28	0.65	0.12	14.00
	4.04	2.19	0.08	1.79	0.31	18.71	0.33	26.83
Shift and Launder	4.16	-	0.22	34.13	5.83	1.92	0.43	40.44
	4.10	2.56	0.14	2.14	0.33	18.73	0.30	27.68
Masked Laundering	4.03	-	0.13	10.90	1.99	1.24	0.63	16.31
	4.28	1.45	0.07	1.10	0.32	9.82	0.61	16.75
Sane Laundering	4.06	-	1.32	35.31	1.03	1.69	0.23	42.40
	4.96	2.39	0.94	2.04	0.18	32.63	0.52	42.99
Recursive Laundering	4.09	-	0.61	4.98	1.49	0.45	0.02	10.14
	4.03	1.63	0.46	6.27	1.64	0.40	0.02	12.81
Call Mask	4.14	-	0.02	0.18	0.15	0.03	0.01	4.38
	4.16	0.57	0.01	0.48	0.19	0.02	0.01	5.25
Dead Recursion	4.10	-	0.10	0.04	0.02	0.00	0.00	4.26
	4.19	0.63	0.01	0.42	0.17	0.01	0.00	5.27

Table 7.3.: The table shows the run times (in ms) for the individual pipeline stages of QIFCI. For every benchmark, the upper row is the execution using no pre-processing and no hybrid analysis, the bottom row is the execution using pre-processing and hybrid analysis. The times given for the hybrid analysis and the execution include the ApproxMC invocations. Additionally, the run time of the model counter is listed separately. The benchmark runs were executed with randomly chosen input arguments. Changing the input arguments does not affect the run time of QIFCI beyond the normal variance.

[7] The tool could execute the analysis until the invocation of ApproxMC for the determination of the dynamic analysis was completed. The ApproxMC output was written to a file in the output directory. QIFCI timed out afterward due to buffering issues.

the inputs values. Input values, that require more loop iterations/recursive calls than the chosen bound allows, are generally treated as belonging to a different indistinguishability set than the examined input. This leads to rather loose approximations where the number of loop iterations of a program execution is often greater than the iteration bound. The effect is visible in the benchmark results of the programs *Parity* and *Masked Laundering*.

Table 7.5 shows the average run time of the benchmark evaluations. The run times are very sensitive to increases in bit width and analysis bounds. The results, however, do not show a dependency of the run time on the specific input for which the dynamic leakage is computed.

The benchmark executions for the *Sum Query* benchmark on 32-bit integers timed out after 30min after the ApproxMC invocation finished. The time-out is most likely the result of buffering issues. We assume the difficulty arises from the high number of models (2^{32}) in combination with the long formulas that are necessary to represent two additions with a total of 96 variables.

7.5. Conclusion

In our evaluation, QIFCI and Nildumu were the only tools that returned safe approximations of the channel capacity for all benchmarked programs. The accuracy of QIFCI (in the configuration without pre-processing and without hybrid analysis) was hereby generally better than Nildumu's: We achieved a better approximation of the channel capacity for the benchmarks *Sanity Check*, *Implicit Flow* and *Sane Laundering* (loop bound = 32). The improvements stem from better handling of implicit flows in QIFCI than in the static analysis that Nildumu uses.

		8 Bit Integers			32 Bit Integers		
	Args	dyn Leak	8	32	dyn Leak	8	32
Masked Copy*	0	4	4	4	16	16	16
Sum Query	0, 1, 2	8	8	8	32	*time out*	*time out*
Sanity Check*	3	0,04	8	8	5.03×10^{-9}	32	32
	1	8	8	8	32	32	32
	17	0.04	0.045	0.045	5.03×10^{-9}	0	0
Table Lookup	0	0.04	0.045	0.045	2.35×10^{-9}	0	0
	1	8	8	8	32	32	32
	17	0.04	0.045	0.045	2.35×10^{-9}	0	0
Implicit Flow	0	0.03	0.045	0.045	2.35×10^{-9}	0	0
	1	8	8	8	32	32	32
	17	0.03	0.045	0.045	2.35×10^{-9}	0	0
Parity*	0	1	1	1	1	32	0.99
Laundering	0	8	8	8	32	32	32
Shift and Launder	0	8	8	8	32	32	32
Masked Laundering	0	8	8	8	32	32	32
	1	1	6	4	1	30	28
Sane Laundering*	0	0.04	0.045	0.045	5.03×10^{-9}	0	0
	1	8	8	8	32	32	32
Recursive Laundering	0	8	8	8	32	32	32
Call Mask	0	4	4	4	28	28	28
Dead Recursion	0	0	0	0	0	0	0

Table 7.4.: The table shows the calculated dynamic leakage for the input arguments shown in the second column. We analyzed each benchmark with different bit widths as well as with different loop and recursion bounds. The column "dynLeak" shows the actual dynamic leakage of the execution. The computations in benchmark programs marked with a * depend on the integer width used in the execution. They have been modified accordingly for the 8-bit integer benchmark runs.

		8 Bit Integers		**32 Bit Integers**	
	Args	8.00	32	8	32
Masked Copy	0	4.48	4.58	4.76	4.61
Sum Query	0, 1, 2	4.90	4.92	*time out*	*time out*
Sanity Check	0	4.64	4.56	6.13	5.41
	1	4.52	4.65	5.91	5.71
	17	4.60	4.74	6.10	5.91
Table Lookup	0	4.76	4.78	5.98	5.68
	1	4.72	4.74	5.44	5.30
	17	4.75	4.76	5.95	5.79
Implicit Flow	0	4.69	4.69	5.21	4.73
	1	4.62	4.67	4.78	4.63
	17	4.62	5.00	4.94	4.67
Parity	0	4.78	4.86	5.17	5.39
Laundering	0	5.06	7.27	7.96	35.89
Shift and Launder	0	5.92	8.41	9.24	49.68
Masked Laundering	0	4.92	6.06	6.71	17.12
	1	4.83	6.15	6.71	17.55
Sane Laundering	0	5.36	7.25	13.63	55.24
	1	5.37	7.42	13.17	54.48
Rec Laundering	0	5.19	6.05	6.61	11.97
Call Mask	0	4.67	4.84	4.75	4.75
Dead Recursion	0	4.66	4.64	4.65	4.69

Table 7.5.: The table shows the average run times (in s) for the dynamic Leakage analysis shown in table 7.4

8. Conclusion and Future Work

The analysis we presented in this thesis can safely approximate the channel capacity and the dynamic leakage of a *while*-program. We have shown that it is generally possible to combine static analysis approaches with SAT-based approaches to improve the run time as well as the accuracy of the analysis.

The effects of the hybrid analysis on the accuracy of the analysis are highly dependent on the input program. While there are input programs that benefit from the hybrid analysis, there are also programs for which the accuracy decreases. The hybrid analysis had a positive effect on the computation time of the analysis, which leads to better scalability of the analysis. Because we tested our analysis on small synthetic benchmarks, it remains open how the hybrid analysis affects the analysis' accuracy for real-world programs.

Additionally to the hybrid analysis of the channel capacity, our tool is able to almost safely approximate the dynamic leakage of a single program execution (under-approximations were minimal if they occurred). We believe this additional data point about the program's information flow is vital in judging the risk of information leakage. It allows the user to identify executions that leak no significant amount of information even if the program would have been deemed as insecure due to a high channel capacity.

Overall we were only able to find a way to combine dynamic and static approaches that benefits a small set of input programs. However, this result can be used as proof that generally, such a combination is possible. The results of this thesis can be used as a basis for more in-depth examinations of the topic.

8.1. Future Work

The analysis and its implementation in their current form leave many areas open for improvements and extensions of the work done so far.

Extending the Analysis Beyond `while`-Programs The input programs of our analysis are currently based on a minimal `while`-language. In particular, we currently lack support for data types apart from integers as well as object-orientation. Memory operations are only minimally supported through the array analysis.

Defining rules for representing these additional program operations as SAT formulas is generally possible, as model checking tools that represent programs as propositional formulas, such as CBMC, already offer support for these features.

Our tool as it is might currently not be scalable enough as especially object orientation and heap operations would require large propositional formulas. Already with the

language in its current scope, the implementation shows scalability issues for more complex input programs and higher loop iteration/recursion bounds.

Easing Restrictions on Input Programs The current analysis expects input programs to adhere to certain restrictions, especially regarding the value that is leaked to the public.

Currently, the program is expected to leak only a single value at the end of the execution. Moving the location of the leak statement to an earlier location in the program does not require any additional work as long as the leak is not contained in a loop or a function call. To handle leaks inside a branch of an `if`-statement, the formulas for the channel capacity and dynamic leakage need to be conjuncted with the execution condition of the block that contains the leak. Allowing a program to leak more than a single value (this includes programs where the leak statement is inside a loop or a function) requires the output to be modeled as a stream, where every location, where the value may have been leaked is taken into consideration.

Increasing Efficiency through Improved Formula Handling The biggest drawback of our implementation compared to the other tools we examined is its lack of scalability. With the increasing size of the formulas that need to be handled, it suffers from major increases in run time. One major reason for the bad scalability is the implementation of the dependency analysis, which requires the program to deal with large formulas. The handling of these large formulas slows down the process of the dependency analysis. Breaking up the large formulas by factoring out individual terms and replacing them with variables offers the possibility of major improvements in the run time.

Improving Accuracy Through Flow Bounds The hybrid approach in the analysis cannot improve the accuracy for input programs where the relation between the different input and output values of the program segments is unclear. An example of this is the benchmark *Masked Laundering*, where the masking segment assumes an input of 33 secret bits. The program, however, only has 32 secret bits.

Flowcheck [16] uses a flow network with edge capacities to limit the flow of information between segments. This prevents the multiplication of input bits as it happens in our tool.

As of now, we were not able to find a way to represent these flow bound restrictions as propositional formulas. It is worth exploring, whether a different way of representing these restrictions (such as in a flow network like Flowcheck does) can be integrated into the analysis.

Adapting Interpreter Actions to Leakage Results Our implementation includes an interpreter that will execute the program and report the dynamic leakage of the execution.

A logical extension would be to adapt the behavior of the interpreter to the calculated leakage. Other tools presented in [17] and [18] adapt the outputs of a program execution

based on the estimated leakage.

By adding a shadow execution with dummy values, whose outputs are used instead of the real ones in cases where the leakage is too high, our tool could be used to execute programs in a safe environment, where the information leakage is guaranteed to be below a defined threshold.

Bibliography

[1] T. Ormandy, "cloudflare: Cloudflare reverse proxies are dumping uninitialized memory." `https://bugs.chromium.org/p/project-zero/issues/detail?id=1139` (accessed: 07.06.2021), 02 2017.

[2] J. Graham-Cumming, "Incident report on memory leak caused by cloudflare parser bug." `https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/` (accessed: 07.06.2021), 02 2017.

[3] D. Denning, "Cryptography and data security," *SERBIULA (sistema Librum 2.0)*, 01 1982.

[4] G. Lowe, "Quantifying information flow," in *Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15*, pp. 18 – 31, 02 2002.

[5] G. Smith, "On the foundations of quantitative information flow," in *Foundations of Software Science and Computational Structures*, pp. 288–302, Springer Berlin Heidelberg, 03 2009.

[6] M. Alvim, K. Chatzikokolakis, A. McIver, C. Morgan, C. Palamidessi, and G. Smith, *The Science of Quantitative Information Flow*. Springer, Cham, 1 ed., 01 2019.

[7] J. Newsome, S. McCamant, and D. Song, "Measuring channel capacity to distinguish undue influence," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, p. 73–85, Association for Computing Machinery, 2009.

[8] V. Klebanov, N. Manthey, and C. Muise, "Sat-based analysis and quantification of information flow in programs," in *QEST*, vol. 8054, pp. 177–192, 08 2013.

[9] D. Kroening and M. Tautschnig, "Cbmc – c bounded model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 389–391, Springer Berlin Heidelberg, 2014.

[10] M. Thurley, "sharpsat – counting models with advanced component caching and implicit bcp," in *Theory and Applications of Satisfiability Testing - SAT 2006*, pp. 424–429, Springer Berlin Heidelberg, 2006.

[11] C. Muise, S. Mcilraith, J. Beck, and E. Hsu, "Dsharp: Fast d-dnnf compilation with sharpsat," in *Proceedings of the 25th Canadian Conference on Advances in Artificial Intelligence*, Canadian AI'12, 05 2012.

[12] F. Biondi, M. A. Enescu, A. Heuser, A. Legay, K. S. Meel, and J. Quilbeuf, "Scalable approximation of quantitative information flow in programs," in *Verification, Model Checking, and Abstract Interpretation* (I. Dillig and J. Palsberg, eds.), (Cham), pp. 71–93, Springer International Publishing, 2018.

[13] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," *ArXiv*, vol. abs/1306.5726, 06 2013.

[14] B. Chu, K. Hashimoto, and H. Seki, "Quantifying dynamic leakage: Complexity analysis and model counting-based calculation," *IEICE Trans. Inf. Syst.*, vol. 102-D, pp. 1952–1965, 2019.

[15] J. Bechberger, "Quantitative information flow control on program dependency graphs," Dec. 2018.

[16] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *PLDI 2008: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pp. 193–205, June 2008.

[17] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *2010 IEEE Symposium on Security and Privacy*, pp. 109–124, 2010.

[18] T. H. Austin, T. Schmitz, and C. Flanagan, "Multiple facets for dynamic information flow with exceptions," *ACM Trans. Program. Lang. Syst.*, vol. 39, May 2017.

[19] A. Bichhawat, *Practical dynamic information flow control.* PhD thesis, Universität des Saarlandes, 2017.

[20] G. Smith, "Principles of secure information flow analysis," in *Malware Detection*, pp. 291–307, Springer US, 01 2007.

[21] N. Bielova, "Dynamic leakage - a need for a new quantitative information flow measure," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, (Vienna, Austria), pp. 83–88, Oct. 2016.

[22] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *2009 30th IEEE Symposium on Security and Privacy*, pp. 141–153, IEEE Computer Society, 2009.

[23] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*, (New York, NY, USA), p. 1–19, Association for Computing Machinery, 1970.

[24] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, p. 12–27, Association for Computing Machinery, 1988.

[25] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," in *International Symposium on Programming* (M. Paul and B. Robinet, eds.), (Berlin, Heidelberg), pp. 125–132, Springer Berlin Heidelberg, 1984.

[26] S. Horwitz, J. Prins, and T. Reps, "On the adequacy of program dependence graphs for representing programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, p. 146–157, Association for Computing Machinery, 1988.

[27] D. Giffhorn and G. Snelting, "Probabilistic noninterference based on program dependence graphs," Tech. Rep. 6, Karlsruhe Institute of Technology, Apr. 2012.

[28] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, p. 439–449, IEEE Press, 1981.

[29] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *Software Development Environments*, 1984.

[30] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, p. 35–46, Association for Computing Machinery, 1988.

[31] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, pp. 399–422, Dec. 2009.

[32] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. NLD: IOS Press, 2009.

[33] L. Valiant, "The complexity of enumeration and reliability problems," *SIAM J. Comput.*, vol. 8, pp. 410–421, 08 1979.

[34] E. Birnbaum and E. L. Lozinskii, "The good old davis-putnam procedure helps counting models," *Journal of Artificial Intelligence Research*, vol. 10, p. 457–477, Jun 1999.

[35] Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[36] A. Darwiche, "Decomposable negation normal form," *J. ACM*, vol. 48, p. 608–647, July 2001.

[37] A. Darwiche, "New advances in compiling cnf to decomposable negation normal form," in *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, (NLD), p. 318–322, IOS Press, 2004.

[38] R. M. Karp, M. Luby, and N. Madras, "Monte-carlo approximation algorithms for enumeration problems," *Journal of Algorithms*, vol. 10, no. 3, pp. 429–448, 1989.

[39] R. Aziz, G. Chu, C. Muise, and P. J. Stuckey, "#∃sat: Projected model counting," in *SAT*, 2015.

[40] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, pp. 399–422, Dec. 2009.

[41] F. Biondi, A. Legay, and J. Quilbeuf, "Comparative analysis of leakage tools on scalable case studies," in *Model Checking Software* (B. Fischer and J. Geldenhuys, eds.), pp. 263–281, Springer International Publishing, 2015.

[42] Z. Meng and G. Smith, "Calculating bounds on information leakage using two-bit patterns," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, (New York, NY, USA), Association for Computing Machinery, 2011.

Erklärung

Hiermit erkläre ich, Tina Maria Strößner, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum

Unterschrift

A. Proof for Equation 2.3

Let p be a deterministic program, with input $\mathtt{H}$ and output $\mathtt{L}$. Let $\mathscr{L}$ be the set of possible outputs.

$$
\begin{aligned}
cc(p) &:= H_\infty(\mathtt{H}) - H_\infty(\mathtt{H} \mid \mathtt{L}) \\
&= H_\infty(\mathtt{H}) - \sum_{l \in \mathscr{L}} P[L = l]\, H_\infty(\mathtt{H} \mid \mathtt{L} = l) \\
&= H_\infty(\mathtt{H}) - \sum_{l \in \mathscr{L}} P[L = l]\, (-L_{dyn}(p, l) + H_\infty(\mathtt{H})) \\
&= H_\infty(\mathtt{H}) - \sum_{l \in \mathscr{L}} P[L = l]\, H_\infty(\mathtt{H}) + \sum_{l \in \mathscr{L}} P[L = l]\, L_{dyn}(p, l) \\
&= \sum_{l \in \mathscr{L}} P[L = l]\, L_{dyn}(p, l) \\
&= \mathbb{E}(L_{dyn}(p, l))
\end{aligned}
$$

The equality in the second line results from the definition of $H(\mathtt{H} \mid \mathtt{L})$, which is given in [5]. All other steps in the proof use the definitions given in chapter 2.1.

B. Proof for Theorem 3.1 and Lemma 3.4

We prove the equivalence theorem and the lemma for the correctness of the execution condition. Both are introduced in section 3:

Theorem (Equivalence Theorem)
Given an input value $h \in \mathscr{H}$ for the program p, let v be an arbitrary value in the program p for which $[\![p]\!]_h(v) \neq \bot$. The relation between the dependency vector and the execution value of v is given by:

$$\forall 0 \leq i < w : \mathcal{V}_h(dVec(v)^i) \iff [\![p]\!]_h(v)^i$$

Lemma (Correctness of $exec(\cdot)$)
For every basic block $b \in \mathrm{BB}_p$ and its execution condition $exec(b)$,
$\mathcal{V}_h(exec(b)) = \texttt{true} \iff$ basic block b is executed in a program run with input h.

Proof Structure We show that the theorem and the lemma are fulfilled for all loop-free programs p that consist of a single function. The proof will be organized as an induction over the basic blocks of the program p. We show the correctness for an arbitrary but fixed input $h \in \mathscr{H}$.

In our induction, we will show that for each basic block of p the following three statements are fulfilled:

H.1 $\big($b is executed in the execution with input $h \implies \mathcal{V}_h(exec(b)) = \texttt{true}\big) \wedge$
$\big($b is not executed in the execution with input $h \implies \mathcal{V}_h(exec(b)) = \texttt{false}\big)$

H.2 For all values defined in b, theorem 3.1 is fulfilled

H.3 If the block b is executed, then for all outgoing edges e of b,
$\mathcal{V}_h(follow(e)) = \texttt{true} \iff$ The execution with input h follows the edge e

Note that statement H.1 is equivalent to the statement made in lemma 3.4.

Assumptions We assume that all expressions in the program p contain at most one operator. The operands of the operator are constant values or variable/parameter accesses. Nested expressions are split: The inner expressions are assigned to new values which in turn are used as the operands for the outer expression.

Base Case

For the base case of the induction, we consider the basic block $b = \texttt{start}$.

Statement H.1 Per definition, $exec(\texttt{start}) = \texttt{true}$. Since the start block is always executed, independent of the input h, H.1 holds.

Statement H.2 The basic block `start` does not contain any statements, hence H.2 holds trivially.

Statement H.3 `Start` has a single successor. For the edge e that connects `start` to its successor, $follow(e) = \texttt{true}$. H.3 holds, since every execution follows the edge e.

Induction Hypothesis

We assume that for an arbitrary basic block $b \in \mathrm{BB}_p$, all blocks b', for which there exists a path from b' to b, fulfill the conditions H.1, H.2, and H.3.

Induction Step

We show that for the basic block $b \in \mathrm{BB}_p$ mentioned in the induction hypothesis, all three conditions H.1, H.2 and H.3 are fulfilled. Remember, that we consider an execution with a fixed input h. We differentiate two cases: (1) the control flow reaches the block b during the execution with input h and (2) the control flow doesn't reach the block b.

Case 1: b is not executed

Statement H.1 Because b is not executed, the first implication of the conjunction is trivially fulfilled. It remains to show that $\mathcal{V}_h(exec(b)) = \texttt{false}$. The formula $exec(b)$ is defined as $exec(b) = \bigvee_{b' \in pred(b)} \big(follow(b', b) \wedge exec(b')\big)$. Thus, $exec(b) = \texttt{false}$, iff for all $b' \in pred(b)$ the condition $follow(b', b) \wedge exec(b')$ evaluates to `false`. This is indeed the case, because for each predecessor b' either

- b' is not executed, therefore $\mathcal{V}_h(exec(b')) = \texttt{false}$ per the induction hypothesis.
- b' is executed therefore $\mathcal{V}_h(exec(b')) = \texttt{true}$ per the induction hypothesis. Because b is not executed the control flow of the execution does not follow the edge (b', b). Per the induction hypothesis for condition H.3, $follow(b', b)$ must evaluate to `false`.

Statement H.2 Since the block b is not executed, for all values v that are defined in b, the execution value $[\![p]\!]_h(v)$ is $\bot$. Because statement H.2 only considers values, for which $[\![p]\!]_h(v) \neq \bot$, H.2 is trivially fulfilled.

Statement H.3 The implication in statement H.3 holds trivially because the premise is not fulfilled.

Case 2: b is executed

Statement H.1 Because b is executed, the second implication of the conjunction is trivially fulfilled. It remains to show that $\mathcal{V}_h(exec(b)) = \texttt{true}$. It suffices to show that there exists at least one $b' \in pred(b)$ for which $\mathcal{V}_h(follow(b', b) \wedge exec(b'))$ is `true`.

We assume that b is executed. This can only be the case if one of b's predecessors is executed and then the control flow is transferred to b. Let b' be this predecessor. Because b' is executed, $\mathcal{V}_h(exec(b')) = \texttt{true}$ per the induction hypothesis. Additionally, since the control flow is transferred along the edge (b', b) and the condition H.3 for b' holds (induction hypothesis), $\mathcal{V}_h(follow(b', b)) = \texttt{true}$.

Statement H.2 The dependency vectors for values in b are computed by evaluating $\mathscr{E}(e)$ for the expression e that defines the value. We prove that statement H.2 holds for all values defined in b by induction of the definition of $\mathscr{E}(\cdot)$.

Let $v \in \text{VAL}_p$ be an arbitrary value defined in b, that is defined by the statement $v \leftarrow e$ and let v_h be the execution value of v.

Base Case

- $e := n, \quad n \in \mathbb{Z} \implies v_h = bv(n)$
  Per definition $\mathscr{E}(e) = bv(n)$. Thus $\forall 0 \leq i < w : \mathcal{V}_h(\mathscr{E}(e))^i \iff v_h^i$

- $e := \texttt{H} \implies v_h = bv(h)$
  Per definition $\mathscr{E}(e) = \mathcal{V}ar(h)$ and $\mathcal{V}_h(\mathcal{V}ar(h)) = bv(h)$. Thus
  $\forall 0 \leq i < w : \mathcal{V}_h(\mathscr{E}(e))^i \iff v_h^i$

Induction Hypothesis Let theorem 3.1 be true for every value v' that appears in the expression e.

Induction Step

- $e := v', \quad v' \in \text{VAL}_p \implies v_h = [\![p]\!]_h(v')$
  For all inputs $h \in \mathscr{H}$, $\mathcal{V}_h(v) = \mathcal{V}_h(v') \overset{I.H.}{=} [\![p]\!]_h(v') \overset{v:=v'}{=} [\![p]\!]_h(v)$

- $e := \hat{e} \oplus \tilde{e}$ or $e := \oplus\hat{e}$ for a bitwise or arithmetic operator $\oplus$
  The function $\mathscr{E}(\cdot)$ is defined by the boolean algebra definitions of the operators. The correctness of the theorem 3.1 follows from the equivalence of the evaluation of boolean algebra and propositional logic.

- $e := \phi(v_0, v_1), \quad v_0, v_1 \in \text{VAL}_p$
  If b contains a ϕ-function with two arguments, then b must have two predecessors. Let b_0 be the first predecessor and b_1 be the second predecessor of b. Then $\mathscr{E}(e)$ is defined as $\mathscr{E}(e) := \mathbb{IF}(exec(b_0), dVec(v_0), dVec(v_1))$.
  - Case $\mathscr{V}_h(exec(b_0)) = \texttt{true}$: From the induction hypothesis of the outer induction, we can conclude that the block b_0 is executed. Thus, in the execution of p, the expression $\phi(v_0, v_1)$ in block b will evaluate to v_0 (because b_0 is the first predecessor of b). Because we assume the program p to be correct, the value v_0 must have been assigned during the execution. Therefore $[\![p]\!]_h(v_0) \neq \bot$ and $[\![p]\!]_h(v) = [\![p]\!]_h(v_0)$. From the induction hypothesis of the inner induction, we can conclude that $\forall 0 \leq i < w : [\![p]\!]_h(v_0)^i \iff \mathscr{V}_h(dVec(v_0)^i)$. From the definition of the ternary operator $\mathbb{IF}(\cdot,\cdot,\cdot)$ and the assumption $\mathscr{V}_h(exec(b_0)) = \texttt{true}$ follows, that the theorem 3.1 is fulfilled for value v.
  - Case $\mathscr{V}_h(exec(b_0)) = \texttt{false}$: From the induction hypothesis of the outer induction, we can conclude that the block b_0 is not executed. Since we assume that b is executed and b only has two predecessors, the second predecessor b_1 must have been executed. We can show that in this case, theorem 3.1 holds using analogous argumentation as in the case $\mathscr{V}_h(exec(b_0)) = \texttt{true}$.
- In the programs we consider, other expressions (for example comparisons) cannot be used to assign a value.

With this, we have shown that statement H.2 is fulfilled for the basic block b.

Statement H.3 In the programs we consider, a basic block can either have zero, one, or two successors.

If the block b has no successors, the statement $H.3$ is trivially fulfilled. If the block b has only one successor b', then that successor is executed if b itself is executed. Per definition $follow(b, b') = \texttt{true}$, thus statement H.3 holds.

Consider the case that b has two successors b_0 and b_1. Let e be the expression that decides to which successor the control flow is transferred. Let without loss of generality, the control flow be transferred to b_0 iff e evaluates to true during the execution.

From the induction hypothesis and the fact that H.2 holds for the block b, we can conclude that all use-values of the expression e fulfill theorem 3.1. The edge annotations $follow(b, b_0)$ and $follow(b, b_1)$ are created by replacing the use-values that appear in e with their dependency vectors. The logical operators remain unchanged. Because the dependency vectors evaluate to the execution values during the execution, the truth value of e doesn't change because of the replacement. Thus, statement H.3 is fulfilled.

We have shown that the conditions H.1, H.2, and H.3 hold for every basic block in the program p if p is executed with the input value h. Since p and h were chosen arbitrarily, the proof holds for every program and every input $h \in \mathscr{H}$. □

C. Benchmark Programs

The benchmarks follow the following conventions: Secret inputs are called h or h1, h2,... if there is more than one input. Public outputs are called l. All public outputs are leaked at the end of the execution.

Small Benchmarks

Masked Copy This benchmark is taken from [42] and modified slightly as the QIFCI tool doesn't support numerical values in binary format. Instead, we create the bitmask using a shift operation. The program masks the 16 highest-value bits from the input and outputs the rest. The channel capacity is 16.

```
int l = h & (-1 << 16);
```

Listing C.1: Masked Copy

Sum Query This benchmark is taken from [22]. The program adds together the three secret input values and returns their sum. The channel capacity is 32 bit.

```
int l = h1 + h2 + h3;
```

Listing C.2: Sum Query

Sanity Check The benchmark was taken from [7]. Because QIFCI and Nildumu are unable to process unsigned integers, we added the condition $0 \Leftarrow h$ to the conditional. We initialized the value `base` with 3. The channel capacity is 4.

```
int l;
if (0 <= h && h < 16) {
    l = 3 + h;
} else {
    l = 3;
}
```

Listing C.3: Sanity Check

Table LookUp This is a variant of benchmark found in [7]. If the input is within the right value range, a value from a pre-initialized array is returned. All other inputs return 0. The channel capacity is 3.

```
int[] table;
table[0] = 0;
table[1] = 1;
table[2] = 2;
table[3] = 3;
table[4] = 4;
table[5] = 5;
table[6] = 6;
table[7] = 7;

int l = 0;

if (0 <= h && h < 8) {
    l = table[h];
} else {
    l= 0;
}
```

Listing C.4: Table LookUp

Implicit Flow This benchmark is a shortened version of an example program given in [7]. The information is leaked via branching. Every if-statement leaks information through an implicit information flow, except for the last one, where the output is assigned its initialization value. The program leaks $\log_2 7 \approx 2,8$ bits of information.

```
int l = 0;
if (h == 1) l = 1;
if (h == 2) l = 2;
if (h == 3) l = 3;
if (h == 4) l = 4;
if (h == 5) l = 5;
if (h == 6) l = 6;
if (h == 7) l = 0;
```

Listing C.5: Implicit Flow

Loop Benchmarks

Parity The program checks whether the input parity has even parity. The difficulty of this benchmark is, that the assignment to in every loop iteration depends on the high input, however, the program can only ever output 0 or 1. The channel capacity is therefore 1.

```
int parity = 0;
int bitSet;

for (int j = 0; j != 32; ++j) {
    bitSet = (h & (1 << j)) != 0 ? 1 : 0;
    parity = (bitSet != parity) ? 1 : 0;
}
int l = parity;
```

Listing C.6: Parity

Laundering Attack The benchmark is taken from [22]. It is a standard example of information being leaked through implicit flows in a loop. All 32 bits of input information are leaked by the program

```
int l = 0;
while (l != h) {
    ++l;
}
```

Listing C.7: Laundering Attack

Shift and Launder This benchmark tests how the tools cope with computations that do not influence the output. The loop computes two values, however, only one is eventually leaked. The channel capacity of the program is 32.

```
int launder = 0;
int shift = 1;
int i = 0;

while (i != h) {
    launder += 1;
    shift = shift << 1;
    i++;
}
int l = launder;
```

Listing C.8: Shift and Launder

Masked Laundering This program first executes a laundering attack but then masks the lowest value bit of the resulting value. The difficulty in this program is, that to recognize that the conditional masks the value, the analysis tool has to correctly identify the relationship between the secret input value and the result of the laundering attack loop. The channel capacity of the program is 31.

```
int l = 0;
while (l != h) {
    l++;
}
if ((h & 1) != 0) {
    l = 1;
}
```

Listing C.9: Masked Laundering

Sane Laundering This program combines the sanity check benchmark with the laundering attack benchmark. The program leaks 4 bits.

```
int x;
if (0 <= h && h < 16) {
    x = 3 + h;
} else {
    x = 3;
}

int l = 0;
for (l != x) {
    ++l;
}
```

Listing C.10: Sane Laundering

Function Calls and Recursion Benchmarks

Recursive Laundering The program executes a laundering attack, however not iteratively in a loop, but through recursion. All 32 bits of information are leaked.

```
int launder(int h, int l) {
    if (h == l) {
        return l;
    }
    return launder(h, l + 1);
}

int l = launder(h, 0);
```

Listing C.11: Recursive Laundering

Call Mask The program masks the four least valued bits of the input value through a chain of function calls. The benchmarks test the ability of the tools to handle nested function calls. The channel capacity of the program is 28.

```
int mask0(int h) {
    return h | (1 << 0);
}

int mask1(int h) {
    return h | (1 << 1);
}

int mask2(int h) {
    return h | (1 << 2);
}

int mask3(int h) {
    return h | (1 << 3);
}

int mask01(int h) {
    return mask1(mask0(h));
}

int mask012(int h) {
    return mask2(mask01(h));
}
```

```
int mask0123(int h) {
    return mask3(mask012(h));
}

int l = mask0123(h)
```

Listing C.12: Call Mask

Dead Recursion The benchmark from [15] contains a recursive function that, depending on the input value, requires many recursive calls, however eventually always returns the same value. The channel capacity of the program is 0.

```
int id(int i) {
    int r = 0;
    if (i > 0) {
        r = id(i - 1) + 1;
    }
    return 0;
}

int l = id(h);
```

Listing C.13: Dead Recursion