Institut für Programmstrukturen
und Datenorganisation (IPD)

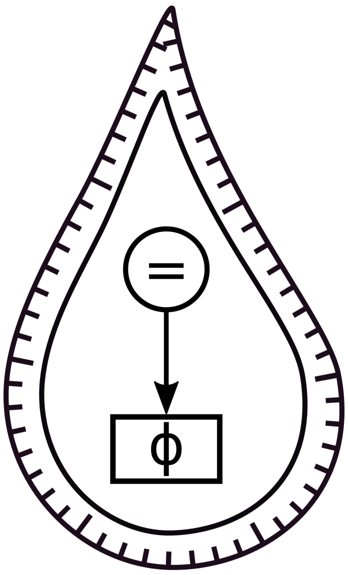
Lehrstuhl Prof. Dr.-Ing. Snelting

Quantitative Information Flow Control on Program Dependency Graphs

Masterarbeit von

Johannes Bechberger

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M.Sc. Simon Bischof

Abgabedatum:	18. Dezember 2018

Abstract

Information flow control is the analysis of the information that an attacker can gain by examining the output of a program. This is a current topic that got more publicity in the recent years as incidents like the HeartBleed bug showed that leaked information can have severe consequences. The HeartBleed bug resulted in the leakage of the secret key for SSL encryption.

The traditional approach in information flow control is qualitative, i.e. examining whether any leakage exists or not. Quantitative Information Flow Control extends this by quantifying the amount of knowledge an attacker can gain on the secret.

This thesis presents a static approach to Quantitative Information Flow Control using a data flow analysis that examines the dependencies between the bits in the program. These dependencies form a graph that is used to calculate an upper bound on the leakage of a program using a minimum-vertex-cut algorithm. The analysis is interprocedural using summary graphs combined with an inlining based approach.

Informationsflusskontrolle ist die Analyse der Information, die ein Angreifer durch die Analyse der Ausgabe eines Programms erlangen kann. Dies ist ein aktuelles Thema, dass in den letzten Jahren immer größere öffentliche Aufmerksamkeit bekam, als Vorfälle wie der HeartBleed-Bug zeigten, dass preisgegebene Informationen schwerwiegende Konsequenzen haben können. Im Falle des HeartBleed-Bugs führte dies zur Preisgabe des geheimen Schlüssels für die SSL-Verschlüsselung.

Der traditionelle Ansatz der Informationsflusskontrolle ist qualitativ. Es wird hierbei untersucht, ob Informationen preisgegeben werden, oder nicht. Quantitativer Informationsfluss erweitert dies mit der Quantifizierung der Menge an Informationen, die ein Angreifer erlangen kann.

Diese Arbeit präsentiert einen statischen Ansatz zur Quantitativen Informationsflusskontrolle, welcher eine Datenflussanalyse nutzt um die Abhängigkeiten zwischen den Bits eines Programmes zu analysieren. Diese Abhängigkeiten bilden einen Graphen, welcher benutzt wird, um eine obere Schranke für die durch ein Programm geleakte Information eines Programms mit Hilfe eines Minimum-Vertex-Cut-Algorithmus zu berechnen. Die präsentierte Analyse ist interprozedural durch die Kombination von Summary-Graphen und einem Inlining-basierten Ansatz.

Contents

1. Introduction

1.1. Motivation

Information flow control deals with the information on the secret inputs leaked by the public outputs of a program.

The traditional approach to information flow control is the qualitative approach. It checks whether secrets can affect the public outputs of a program in any way and are thereby leaked to an attacker.

```
o = h % 10
```

Listing 1.1: Program that explicitly leaks the last digit of the secret input

An example for a leak is given in the program presented in Listing 1.1, with h being the secret input and o the public output. This program explicitly leaks the last digit of the secret. An implicit leak can be seen in the program given in Listing 1.2, called *Password checker*. This program returns 1 if the secret input is equal to the public, attacker-chosen, input l. Both leaks can be found by Qualitative Information Flow Control. This has many applications ranging from the security of distributed applications to formally certifying data storage systems [1].

Qualitative Information Flow Control is an established area of research that produced tools like JOANA [2] that scale to 100K lines of real-world Java code. The problem is that small leaks may often be acceptable and sometimes necessary; the amount of the leaked information is therefore necessary to know. In the case of the *Password checker* program, the leakage of the information whether the secret has a specific value is necessary.

```
if (h == l) {
  o = 1
} else {
  o = 0
}
```

Listing 1.2: Password checker program

Qualitative Information Flow Control cannot distinguish between this program and the program that leaks the whole secret. The urge to distinguish between such cases leads to the need of quantifying the leakage. The aim of Quantitative Information Flow Control is to calculate the leakage of a program in bits. The quantified leakage of the *Password checker* is clearly lower than the leakage of the program that leaks the whole secret. Quantitative Information Flow analyses are called *sound* if the analysis returns an upper bound on the leakage.

Many analyses were proposed in recent years, a few of them can analyse the leakage of real-world programs, see Appendix B. The problem with most of these analyses is that they are at most theoretically sound or do not support functions. There are essentially three classes of analyses:

Static The program is analysed without executing it. These analyses have the advantage of being sound. Most papers focus on type systems, abstract semantics, statistical models or bit variability patterns. A subclass is the class of theoretically sound analyses.

Theoretically Sound Static These analyses only work theoretically for unbounded finite loops and recursion. In practice, only a limited recursion depth and a limited number of loop iterations can be soundly analysed due to resource limitations.

Probabilistic Static The program is analysed statically, but the analysis only gives a confidence interval for the upper bound of the leakage.

Dynamic The leakage is approximated by executing the program with different inputs. Such analyses can only approximate the leakage of programs without giving upper bounds for the leakage. The reason for this is the infeasibility to execute a program with all possible combinations of inputs.

We found no analysis that uses data flow techniques, which are commonly used in the field of compiler development. Furthermore, there are, to our knowledge, no sound analyses that support unbounded finite loops and recursion, see Appendix B.

This thesis presents such an analysis. The analysis is based on a constant bit analysis [3].

1.2. Related Work

Quantitative Information Flow has first been formally defined by Lowe [4] as an extension of information flow that quantifies the information passed from a user on a higher level to a user on a lower level. The theoretical foundations are described by Smith [5]. Smith describes the min-entropy measure and other general concepts of Quantitative Information Flow analyses. The analyses can be categorised in different classes, as described before. The following gives a short overview, a comprehensive survey on analysis tools is presented in Appendix B:

Static analyses are typically based on program algebras, model counting or abstract interpretation. An algebraic framework has been presented by Malacaria [6]. One of the first model counting analyses has been developed by Newsome et al. [7] and abstract interpretations are summarised by Smith [8]. Recent advances in the field of approximative model counting resulted in the development of analyses that can process real-world code; such an analysis tool, *ApproxFlow* [9], is evaluated in Chapter 9. An example for a recent analysis based on abstract interpretation is *jpf-qif* developed by Phan et al. [10]. The *jpf-quilura* analysis [11] by the same authors combines abstract interpretation and model counting using reliability analysis.

Dynamic analyses are typically based on statistical sampling or dynamic tainting. Tainting describes the process of marking values if they are influenced by the secret input. An analysis based on statistical sampling is *LeakWatch* [12]. Rather recently, tools like *Kite* [13] explored the combination of dynamic tainting with model counting. The advantage of the dynamic approaches is that they work on thousands of lines of code, producing precise leakage estimations for a limited input space.

Furthermore, although there is no other data flow based static analysis, there are, to our knowledge, two analyses based on the same ideas as this thesis: The dynamic analysis described by McCamant et al. [14] uses a dynamic tainting based approach instead of statically tracking the flow of information through the program. The static analysis by Mu [15] tracks the dependencies between variables in a PDG representation of the program and computes the value distributions for each program variable. The analysis is able to soundly analyse programs of a while-language using a probabilistic denotational semantic.

2. Foundations

This chapter describes the theoretical foundations of this thesis. It covers the foundations of Quantitative Information Flow, Program Dependency Graphs and the underlying constant bit analysis.

2.1. Quantitative Information Flow

Quantitative Information Flow is formalised in this section using an information theoretical notion of leakage [5]. This is based on Shannon's concept of entropy [16]. The entropy is the amount of information a specific value of a random variable conveys, given in bit. In the following, `H` is the secret input, `O` the public output and $H(\cdot)$ a measure of entropy.

Definition 2.1 (Leakage)**.** The leakage is the mutual information of `H` and `O`, as defined by the following equation:

$$\underbrace{I(\mathtt{H};\mathtt{O})}_{\text{leakage}} = \underbrace{H(\mathtt{H})}_{\text{initial uncertainty}} - \underbrace{H(\mathtt{H}|\mathtt{O})}_{\text{remaining uncertainty}}$$

$H(\mathtt{H})$ is the amount of information that the secret `H` conveys. $H(\mathtt{H}|\mathtt{O})$ is the amount of uncertainty on the value of `H` after observing the program's public output `O`. The mutual information then gives the amount of information on `H` that is leaked to the observer of `O`.

One of the typically used entropy measures for leakage computation is the *min-entropy*.

Definition 2.2 (Min-entropy)**.** Min-entropy H_∞ is based on the concept of vulnerability [5]:

$$V(\mathtt{X}) = \max_{x \in \mathcal{X}} P[\mathtt{X} = x]$$
$$H_\infty(\mathtt{X}) = \log_2 \frac{1}{V(\mathtt{X})}$$

Min-entropy is related to the probability of a secret being guessed in one try, as $V(\mathtt{X})$ is the worst-case probability that `X`'s value can be guessed correctly in one try.

Theorem 2.1 (Min-entropy for deterministic programs)**.** *The min-entropy leakage can be calculated for deterministic programs with a uniform distribution of secrets by* *counting the different possible outputs of a program [5]. Let O be the set of possible outputs of the program, then:*

$$I_\infty(\mathtt{H};\mathtt{O}) = H_\infty(\mathtt{H}) - H_\infty(\mathtt{H}|\mathtt{O}) = \log_2 |O|$$

The calculated leakage is also the maximum leakage over all distributions of secrets, therefore the calculated min-entropy can be seen as the capacity of the program when used as an information theoretic channel.

Attacker model We assume that attackers have access to the program code, can set low inputs at the beginning and see low outputs after program termination. More general, for arbitrary security lattices, an attacker on security level α can see all outputs with level $\leq \alpha$ and set and see inputs with level $\leq \alpha$. All other inputs and outputs are inaccessible. The runtime of the program, other side channels like power consumption, or intermediate states of variables in the program cannot be observed.

Security lattice The different security levels, like “low” ($\hat{l}$) and “high” ($\hat{h}$), can be organised in a lattice that is typically bounded and complete [17]. A security lattice gives the different levels an ordering.

Security lattice The different security levels, like “low” ($\hat{l}$) and “high” ($\hat{h}$), can be organised in a lattice that is typically bounded and complete [17]. A security lattice gives the different levels a partial ordering.

Assumptions on analysed programs All analysed programs are deterministic and terminate normally, i.e. without an exception. The programs are assumed to be written in a variant of a while-language with functions and recursion.

2.2. Program Dependency Graphs

Programs can be represented in varying ways, but there is a consensus that Program Dependency Graphs (PDG) are valuable for program analyses [18] and that they can be used for information flow analyses [19]. This thesis uses PDGs as its main representation of programs.

PDGs were first introduced by Ferrante et al. [20]. Each node in the PDG represents a statement or a part of it. There are two edge types: control dependency and data dependency edges. A statement `A` control-depends on another statement `B` if the execution or non-execution of `A` depends on the result of the execution of `B` and there is no other statement `C` that control depends on `B` and `A` control depends on. `B` can be seen as the conditional control flow statement whose control flow structure directly encloses `A`.

There are different variants of PDGs, depending on the form of the program statements. The basic variant is based on the Single Static Assignment (SSA) form,

```
if (h == l) {
  o1 = 1
} else {
  o2 = 0
}
o = φ(o1, o2)
```

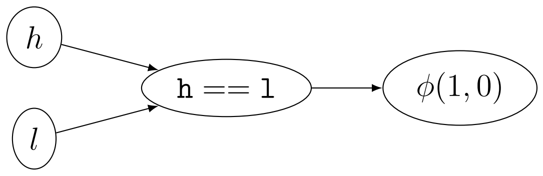


Figure 2.1.: SSA-form and basic PDG for the Password checker program Listing 9.8

which is an intermediate form of a program in which each variable is only assigned once [18, p. 252f]. This makes the relation between variable definition and usage explicit. ϕ-functions are introduced during the transformation into SSA-form to join the different definitions of the same variable coming from different branches of control structures. An example for a SSA-PDG is given in Figure 2.1. The SSA variant used in this thesis is called Gated Single Static Assignment (GSSA) [21]. In GSSA-form the ϕ-functions are annotated with conditions that determine which of the incoming variables are actually used depending on the execution context.

PDGs can be extended to System Dependency Graphs (SDG) [22] that support functions. SDGs and PDGs are often used interchangeably as is it clear from the context whether the analysis is intra- or interprocedural.

2.3. Constant Bit Analysis

The aim of the constant bit analysis is to find bits that are statically known. This section explains the basic elements of the analysis described by Budiu et al. [3].

The core of the analysis is a data flow analysis that combines a constant propagation with an unreachable code elimination, to analyse only reachable portions of the program. A basic combination of these analyses was first presented by Wegman et al. [23] and formalised by Click et al. [24]. The combined analysis presented here works on PDGs and associates each PDG node with a bit string value.

Definition 2.3 (Value lattice)**.** Integer values are represented as bit strings in two's complement [25]. The values have a fixed width n for brevity. Boolean values are represented as $\underbrace{0\ldots0}_{n-1}1$ (*true*) and $\underbrace{0\ldots0}_{n-1}0$ (*false*). The value lattice is a product lattice of bits:

$$\mathit{Value} = \underbrace{\mathit{Bit} \times \cdots \times \mathit{Bit}}_{n \text{ bits}}$$

In the basic version of this analysis, the bit lattice is equivalent to the bit value lattice defined in the following.

Definition 2.4 (Bit value lattice)**.** Bits are represented by elements of a bit value lattice $\mathcal{B}$, its structure is presented in Figure 2.2:

$$\begin{tikzcd}
 & \hat{u} = \top & \\
\hat{0} \arrow[ur] & & \hat{1} \arrow[ul] \\
 & \hat{x} = \bot \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 2.2.: Structure of the bit lattice

Each bit can be represented by one of the following *Bit* lattice elements:

$\hat{u}$ The value of the bit is statically unknown.

$\hat{0}$, $\hat{1}$ The value of the bit is statically known to be either 0 or 1.

$\hat{x}$ The analysis did not yet evaluate the PDG node that this bit belongs to.

The focus of the security analysis is on statically unknown bits, as they might leak information to an attacker.

3. Basic Analysis

This chapter explains the basic structure of the Quantitative Information Flow analysis for loop-free programs. The analysis directly builds upon the constant bit analysis. This chapter first describes the basic idea, then the program analysis and finally three basic operators.

3.1. Basic Idea

The analysis extends the constant bit analysis by tracking all dependencies between the bits of different nodes. The resulting bit dependency graph will be analysed to approximate the leakage of a program, as described in Chapter 4.

3.1.1. Bit and Value lattice

The aim to track the dependencies results in an extension of the original bit lattice.

Definition 3.1 (Bit lattice).

$$Bit = \underbrace{\mathcal{B}}_{\text{bit value } = \text{ v}} \times \underbrace{\mathcal{P}(Identity)}_{\text{dependencies } = \text{ d}}$$

The bit lattice is a product lattice consisting of the bit value lattice *(Definition 2.4)* and a power set lattice containing the data and control dependencies of each bit.

Definition 3.2 (Identity). The identity of a bit consists of the PDG node with which it is associated and its index in the bit string:

$$Identity = Node \times \underbrace{\mathbb{N}_0}_{index} .$$

Definition 3.3 (Value). The values of the nodes are modelled as:

$$Value = \underbrace{Identity \times \cdots \times Identity}_{\text{n bits}}$$

Definition 3.4 (*bitMap* mapping). The *bitMap* maps a bit's identity to its current value in the analysis:

$$bitMap : Identity \to Bit$$

Definition 3.5 (*nodeValue* function)**.** The *nodeValue* function maps a PDG node to its value:

$$\begin{aligned} \mathit{nodeValue} : \mathit{Node} &\to \mathit{Value} \\ x &\mapsto \big((x,\ n-1),\ \ldots,\ (x,\ 1),\ (x,\ 0)\big) \end{aligned}$$

The following text uses bits and their identities interchangeably. Bits are often referenced by their identity using the notation $\mathit{node}^{\mathit{index}}$. This implicitly uses the *bitMap* mapping and the *nodeValue* function:

$$\mathit{node}^{\mathit{index}} \equiv \mathit{bitMap}\big(\mathit{nodeValue}(\mathit{node})^{\mathit{index}}\big)$$

Definition 3.6 (Input bits)**.** Input bits belong to the input values of the program and therefore have a statically unknown value and no dependencies. They have an inherent security level.

Definition 3.7 (Inherent security level)**.** The function sec : $\mathit{Identity} \to S$ maps inputs bit to their inherent security level and all other bits with no such level to the lowest level.

$$\mathrm{sec} : \mathit{Identity} \to S, b \mapsto \begin{cases} \alpha & \text{b is an input of level } \alpha \\ \hat{l} & \text{else} \end{cases}$$

Definition 3.8 (Output bits)**.** Output bits are the bits assigned to output variables of a specific security level.

3.1.2. Notation for Bits and Values

Elements of the bit and value lattice are often used in the following sections and chapters, their notation is given in the following.

A value is notated as a tuple of its bits:

$$(b^{n-1}, \ldots, b^0)$$

The notation for a bit itself is:

$$\Big(\underbrace{[\hat{x}|\hat{0}|\hat{1}|\hat{u}]}_{\text{bit value}},\ \underbrace{\{id1, \ldots, id\gamma\}}_{\text{dependencies}}\Big)$$

An example is the constant bit 0: $(\hat{0}, \varnothing)$. A short notation for the bit value of the ith bit of a value or node x is v_x^i, an analogous notation is used for dependencies d.

```
def evaluate(node: Node):
  args = [nodeValue(p) for p in paramNode(node)]
  for i, new in enumerate(op(node)(args)):
    bitMap(node^i) = bitMap(new)
```

Listing 3.1: Basic transfer function

3.2. Program Analysis

This section describes the analysis of loop-free programs based on the constant bit data flow analysis. During this analysis, the bits associated with the value of each PDG node are stored in the *bitMap* mapping.

Definition 3.9 (Operator)**.** An operator formalises the effect of a PDG node on its arguments:

$$\mathit{Operator} = \mathit{Seq[Value]} \to \mathit{Value}$$

This uses the notation *Seq[τ]* for a finite sequence of elements of τ.

The function *op* returns the operator function for each node:

$$\mathit{op} : \mathit{Node} \to (\mathit{Seq[Value]} \to \mathit{Value})$$

The function *paramNode* returns the sequence of data dependencies that constitute the parameter nodes:

$$\mathit{paramNode} : \mathit{Node} \to \underbrace{\mathit{Seq[Node]}}_{\text{parameter nodes}}$$

Definition 3.10 (*evaluate* function)**.** The *evaluate* function is the transfer function of the data flow analysis. This function is defined in Listing 3.1. It stores the result of the operator application into the *bitMap* mapping.

Tracking the dependencies Dependencies are tracked by the operators that process bits. The control dependencies can be gathered from annotations of the GSSA-PDG, using the function *incomingControlDeps*.

3.3. Operator Specification

In the following, we specify three operators that we need for meaningful examples in the next chapter; a larger set is specified in Section 5.2.

In the following, x is the first operand, y the second operand and r the result of each operator.

The equality operator

All result bits besides the least significant bit have the constant value $\hat{0}$ as the result of the operator is of type boolean.

$$r^i = (\hat{0}, \varnothing) \qquad 0 < i < n$$

The bit value of the least significant result bit is $\hat{1}$ if both operands have the same constant value. The bit value is $\hat{0}$ if both operands have different constant values at one index and $\hat{u}$ otherwise:

$$v_r^0 = \begin{cases} \hat{1} & \forall i : v_x^i = v_y^i \wedge v_x^i, v_y^i \in \{\hat{0}, \hat{1}\} \\ \hat{0} & \exists i : v_x^i \neq v_y^i \wedge v_x^i, v_y^i \in \{\hat{0}, \hat{1}\} \\ \hat{u} & \text{else} \end{cases}$$

The least significant result bit depends on all statically unknown bits of both operands if the result is statically unknown:

$$d_r^0 = \begin{cases} \{\alpha^i | \alpha \in \{x, y\}, v_\alpha^i = \hat{u}, 0 \leq i < n\} & v_r^0 = \hat{u} \\ \varnothing & \text{else} \end{cases}$$

The bitwise or-operator The operator can be defined bitwise:

$$v_r^i = \begin{cases} \hat{1} & v_x^i = \hat{1} \vee v_y^i = \hat{1} \\ \hat{0} & v_x^i = v_y^i = \hat{0} \\ \hat{u} & \text{else} \end{cases} \qquad 0 \leq i < n$$

Each statically unknown result bit depends on the respective operand bits that are statically unknown at index i:

$$d_r^i = \begin{cases} \{\alpha^i | \alpha \in \{x, y\}, v_\alpha^i = \hat{u}\} & v_r^i = \hat{u} \\ \varnothing & \text{else} \end{cases}$$

The ϕ-operator This operator works with an arbitrary number of values α_j in a bitwise manner. The bit value of the result bit is the supremum of the bit values of all operand bits.

$$v_r^i = \bigsqcup_j v_{\alpha_j}^i \qquad 0 \leq i < n$$

The control dependencies are gathered by considering the values of the nodes that the ϕ-node control-depends on. The helper function $cs : \mathit{Node}_\phi \to \mathcal{P}(\mathit{Identity})$ returns the control dependencies for each ϕ-node φ:

$$cs(\varphi) = \{\mu^0 | \mu \in \mathit{incomingControlDeps}(\varphi), v_\mu^0 = \hat{u}\}$$

The control dependencies are combined with the data dependencies to form the dependencies of the result bit. We assume that φ is the current ϕ-node.

$$d_r^i = \begin{cases} \{\alpha^i | v_\alpha^i = \hat{u}, 0 \leq i < n\} \cup cs(\varphi) & v_r^i = \hat{u} \\ \varnothing & \text{else} \end{cases}$$

4. Leakage Computation

This chapter explains the sound approximation of the leakage of a program using the bit dependency graphs created by the program analysis of the previous chapter.

The leakage computation consists of calculating the maximum number of different outputs over all low inputs *(Theorem 2.1)*. The logarithm of this number is an upper bound on the leakage of the program [5]. In the following, the low inputs are assumed to be arbitrary, but fixed.

Definition 4.1 (Bit dependency graph)**.** The bits and their dependencies form the bit dependency graph. This is a directed graph. Nodes of this graph represent the bits. This graph contains an edge from a bit a to a bit b if b depends on a.

Definition 4.2 (Leakage calculation function)**.** The leakage calculation can be formalised as a function

$$C : \mathit{Value} \to \mathbb{R}_{\geq 0}$$

that returns the leakage of a program for its output value. The output value is a result of the analysis presented in the previous chapter. The program itself is omitted in the function definition for brevity. In the following, the result of the leakage computation is in $\mathbb{N}_0$.

4.1. Basic Approximations

There are two basic approximations for the leakage calculation that are described in the following.

4.1.1. Counting high input bits

The first basic leakage approximation C_{input} is calculated by counting all high input bits that the output depends on. The set of input bits that a set of bits transitively depends on can be calculated recursively:

$$\begin{aligned} \mathit{inputBits} : \mathcal{P}(\mathit{Identity}) &\to \mathcal{P}(\mathit{Identity}) \\ bs &\mapsto \bigcup_{b \in bs} \begin{cases} \{b\} & b \text{ is an input bit} \\ \mathit{inputBits}(d_b) & \text{else} \end{cases} \end{aligned}$$

The high input bits that a set of bits depends on can then be calculated using the *sec* mapping:

$$\begin{aligned} \mathit{highInputBits} : \mathcal{P}(\mathit{Identity}) &\to \mathcal{P}(\mathit{Identity}) \\ \mathit{bs} &\mapsto \{b | b \in \mathit{inputBits}(\mathit{bs}), \mathit{sec}(b) = \hat{h}\} \end{aligned}$$

Finally C_{input} can be defined as:

$$\begin{aligned} C_{\mathit{input}} : \mathit{Value} &\to \mathbb{N}_0 \\ \mathit{val} &\mapsto |\mathit{highInputBits}(\mathit{val})| \end{aligned}$$

This leakage approximation is sound because the number of different outputs is bounded by the number of different input bits that can influence the outputs, as the analysed program is deterministic and PDGs include all dependencies between the inputs and outputs.

4.1.2. Counting high dependent output bits

Another basic approximation is to count the number of high dependent bits in the output value. Using the previous basic approximation, a bit b can be defined as depending on a high input bit if $C_{\mathit{input}}\big((b)\big) > 0$. This results in the leakage approximation C_{output}:

$$\begin{aligned} C_{\mathit{output}} : \mathit{Value} &\to \mathbb{N}_0 \\ \mathit{val} &\mapsto |\{b | b \in \mathit{val}, C_{\mathit{input}}\big((b)\big) > 0\}| \end{aligned}$$

This is a direct result of the observation that an output bit whose value is not influenced by a high input variable cannot leak any secret information. The logarithm of the number of different outputs is upper-bounded by the number of unknown output bits, as each unknown bit doubles the number of possible outputs. Only high dependent bits are considered, as the analysis assumes that all low input bits are arbitrary but fixed.

4.1.3. Combined basic approximation

The two basic leakage approximations are both sound over-approximations, therefore using the minimum leakage calculated by both is a sound approximation too:

$$\begin{aligned} C : \mathit{Value} &\to \mathbb{N}_0 \\ \mathit{val} &\mapsto \min\big(C_{\mathit{input}}(\mathit{val}), C_{\mathit{output}}(\mathit{val})\big) \end{aligned}$$

4.1.4. Examples

The following examples show how to calculate the leakage for basic programs. All examples use two bit values, $h = (h^1,\ h^0)$ as high input, $l = (l^1,\ l^0)$ as low input and $o = (o^1,\ o^0)$ as low output for simplicity.

```
o = h == 0
```

Listing 4.1: Program with a basic comparison

```
1 if (h == 0) {
2   o1 = 0b00
3 } else {
4   o2 = 0b11
5 }
6 o = ϕ(o1,o2)
```

Listing 4.2: Program with a basic if-statement

Program with basic comparison The program given in Listing 4.1 uses the equality operator. The least significant bit of the result is 0 if h has the value 0, i.e. $v_h^0 = \hat{0} \wedge v_h^1 = \hat{0}$, and 0 otherwise. The least significant bit of o is therefore data dependent on both bits of h, as the value of the secret is statically unknown: $o = \big((\hat{0}, \varnothing),\ (\hat{u},\ \{h^1, h^0\})\big)$. The leakage of the program is 1, as one output bit depends on two input bits.

$$C(o) = \min\left(C_{input}(o), C_{output}(o)\right) = \min(2, 1) = 1$$

Program with basic if-statement The following program gives an example for a program with an if-statement, that uses the comparison expression of the previous example as a condition. Its code is presented in Listing 4.2.

The condition `h == 0` evaluates to $\big((\hat{0}, \varnothing),\ (\hat{u},\ \{h^1, h^0\})\big)$ as the result of the comparison is unknown. The variable `o1` has the value $\big((\hat{0}, \varnothing),\ (\hat{0}, \varnothing)\big)$, the variable `o2` the value $\big((\hat{1}, \varnothing),\ (\hat{1}, \varnothing)\big)$. The ϕ-operator in line 5 has the arguments `o1` and `o2`. The result of evaluating the ϕ-node is $o = \big((\hat{u},\ \{\texttt{h == 0})^0\}),\ (\hat{u},\ \{(\texttt{h == 0})^0\})\big)$ as $v_{o1}^0 \neq v_{o2}^0$, $v_{o1}^1 \neq v_{o2}^1$ and the node depends on the condition `h == 0`. The output `o` therefore depends on two high input bits and consists of two unknown bits that depend on high input bits. As a result the leakage is approximated to be

$$C(o) = \min(2, 2) = 2.$$

4.2. Generalised Leakage Approximation

The aim of this section is to generalise the basic leakage approximations, so that they take the structure of the bit dependency graph into account. This section starts with some key observations, followed by a concrete algorithm.

A first observation is that we can approximate the leakage conservatively by either considering a bit b itself, counting it as one if $C_{input}\big((b)\big) > 0$, or by combining the leakage approximated for all bits b depends on. In other words: Instead of considering b as high dependent, we can also count the high dependent bits that b depends on. This is because the value of b is fixed if we fix all the bit's dependencies. The idea is that there are often overlaps between the bits that different bits depend on, thereby reducing the number of bits to consider and fix for an output value consisting of multiple bits. The goal is to find the minimal set of bits that have to be fixed to fix the output, assuming that low input and constant bits are considered as fixed by default and can be ignored. The discovered set of bits fully defines the output.

This leads to an optimisation problem which can be transformed into instances of different well-known optimisation problems for which solvers already exist. The aim it to use these solvers to approximate the leakage. The following section describes how the leakage approximation can be directly transformed into a minimum-vertex-cut problem. Appendix A contains a transformation into instances of another optimisation problem called Partial MAXSAT.

Example The following is a reevaluation of the short example given in Listing 4.2. This shows how the leakage approximation works and that it is more precise than the previous approximations.

The analysis of the program works as before, it computes the following values:

$$\begin{aligned}
(\texttt{h} == \texttt{0}) &= \big((\hat{0}, \varnothing),\ (\hat{u},\ \{h^1, h^0\})\big)\\
o1 &= \big((\hat{0}, \varnothing),\ (\hat{0}, \varnothing)\big)\\
o2 &= \big((\hat{1}, \varnothing),\ (\hat{1}, \varnothing)\big)\\
o &= \big((\hat{u},\ \{(\texttt{h} == \texttt{0})^0\}),\ (\hat{u},\ \{(\texttt{h} == \texttt{0})^0\})\big)
\end{aligned}$$

Fixing the bit $(\texttt{h} == \texttt{0})^0$ is enough to fix the output, as both output bits only depend on $(\texttt{h} == \texttt{0})^0$. The leakage of the program is therefore 1. Fixing the output bits o^1 and o^0 would also fix the output, but this set of fixed bits would not be minimal, as fixing the single bit $(\texttt{h} == \texttt{0})^0$ is enough. The same holds for the input bits. The basic leakage approximations calculate a leakage of 2. This is an example for the precision improvement gained using the generalised leakage approximation.

4.2.1. Leakage approximation using Minimum-Vertex-Cut

We extend the bit dependency graph *(Definition 4.1)* to include an input and an output pseudo-vertex. All output bits have an edge to the output pseudo-vertex. The input pseudo-vertex has an edge to all high input bits. The secret information can be seen as flowing from the input pseudo-vertex to the output pseudo-vertex. Every bit vertex has weight 1 as each bit can leak at most one bit. This results in a transformation of the information flow problem into a maximum flow problem, so we can use pre-existing solving algorithms. An example for a leakage graph is given in Figure 4.1.

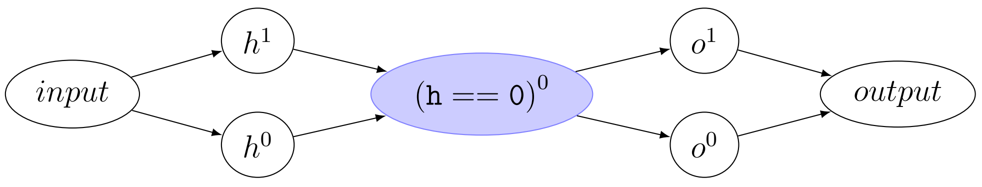


Figure 4.1.: The leakage graph for Listing 4.2

The leakage can then be approximated by calculating the minimum-vertex-cut through the graph. The minimum-vertex-cut of a graph is the minimal set of vertices that need to be removed to cut all connections between the input and the output pseudo-vertex [26]. This is equivalent to finding the minimal set of bits that need to be fixed to fix the output bits. In the example above, the minimum-vertex-cut is calculated as $\{(\texttt{h} == \texttt{0})^0\}$, resulting in a calculated leakage of 1, as expected.

There are many algorithms for finding the minimum-vertex-cut in polynomial time, a comprehensive survey was created by Esfahanian [27]. The most common approach is to transform the minimum-vertex-cut problem into a minimum-cut problem [28, p. 122f]. The minimum-cut of a graph is the minimal set of edges that need to be removed from the graph to cut all connections between the input and the output pseudo-vertex.

The minimum-vertex-cut based leakage approximation is correct by construction as the bits in the vertex-cut constitute a set of bits that defines the output. Every transitive dependency of the output to the input goes through at least one of the bits in the vertex-cut. Furthermore the leakage approximation is precise as the minimum-vertex-cut constitutes a minimal set of bits that have to be fixed.

Asymptotic runtime of the computation During the transformation, each node is split into two, connected by an edge with weight 1; all other edges have an infinite weight. This results in an edge-weighted-graph. The Ford-Fulkerson algorithm has an asymptotic runtime of $\mathcal{O}(|\mathit{edges}| \cdot \mathit{maxflow})$ [29]. Assuming that the arity of each operator is upper-bounded and values are w bits wide, a single bit can only depend on $\mathcal{O}(w)$ other bits, resulting in $\mathcal{O}(w \cdot |\mathit{bits}|)$ edges. The leakage of a program is bounded both by the number of output and by the number of input bits. The runtime of Ford-Fulkerson can therefore be given as $\mathcal{O}\big(w \cdot |\mathit{bits}| \cdot \min(|\mathit{input\ bits}|, |\mathit{output\ bits}|)\big)$.

Optimisation It is possible to adapt minimum-cut algorithms to work on vertex weighted graphs, erasing the need for the construction of a new graph data structure. The idea is to take advantage of the fact that all outer-edges have an infinite weight.

4.2.2. Example for using the Minimum-Vertex-Cut approach

The following example, given in Listing 4.3, is more complex than the previous ones and uses 3 bit values. It uses the notation α`[`i`] == 1` to express that only the i^{th} bit of α is used for the comparison. The $\cdot[\cdot]$ operator is emulatable using the bitwise

```
z = h[1] == 1
if (h[2] == 1) {
  if (z){
    o11 = 0b000
  } else {
    o12 = 0b001
  }
  o1 = ϕ(o11, o12)
} else {
  o2 = 0b111
}
o = ϕ(o1, o2)
```

Listing 4.3: Example with nested if-statements

and-operator and the shift operators.
The analysis of the program works as before. The following values are gathered by the analysis:

$$\begin{aligned} z &= \big((\hat{0}, \varnothing),\ (\hat{0}, \varnothing),\ (\hat{u},\ \{h^1\})\big) \\ (\mathtt{h}[2] == \mathtt{1}) &= \big((\hat{0}, \varnothing),\ (\hat{0}, \varnothing),\ (\hat{u},\ \{h^2\})\big) \\ o11 &= \big((\hat{0}, \varnothing),\ (\hat{0}, \varnothing),\ (\hat{0}, \varnothing)\big) \\ o12 &= \big((\hat{0}, \varnothing),\ (\hat{0}, \varnothing),\ (\hat{1}, \varnothing)\big) \\ o1 &= \big((\hat{0}, \varnothing),\ (\hat{0}, \varnothing),\ (\hat{u},\ \{z^0\})\big) \\ o2 &= \big((\hat{1}, \varnothing),\ (\hat{1}, \varnothing),\ (\hat{1}, \varnothing)\big) \end{aligned}$$

The last ϕ-node then evaluates to:

$$o = \big((\hat{u},\ \{(\mathtt{h}[2] == \mathtt{1})^0\}),\ (\hat{u},\ \{(\mathtt{h}[2] == \mathtt{1})^0\}),\ (\hat{u},\ \{(\mathtt{h}[2] == \mathtt{1})^0, o1^0\})\big)$$

The bit dependency graph is then transformed into a graph for minimum-vertex-cut problem, given in Figure 4.2. The minimum-vertex-cut consists of z^0 and $(\mathtt{h}[2] == \mathtt{1})^0$, as a result the leakage is approximated as 2 which is a precise upper integer bound of the real leakage of $\log_2 3$ as the example program can have three different outputs.

4.2.3. Example with n-bit values

We give in the following an example where the generalised leakage approximation computes a constant and the basic approximations an arbitrary leakage, depending on the width of the values in the program. The example code is presented in Listing 4.4.

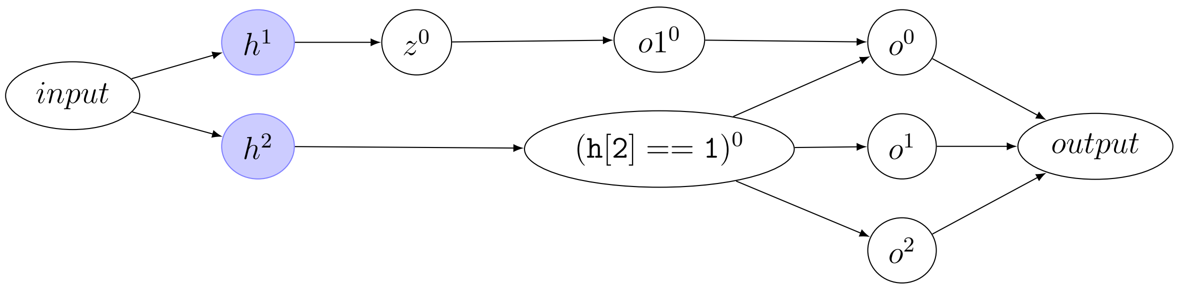


Figure 4.2.: Bit dependency graph for Listing 4.3.

```
if (h == 0) {
  o1 = 0b000...0        // n 0s
} else {
  o2 = 0b111...1        // n 1s
}
o = ϕ(o1, o2)
```

Listing 4.4: If-statement example with n bit values

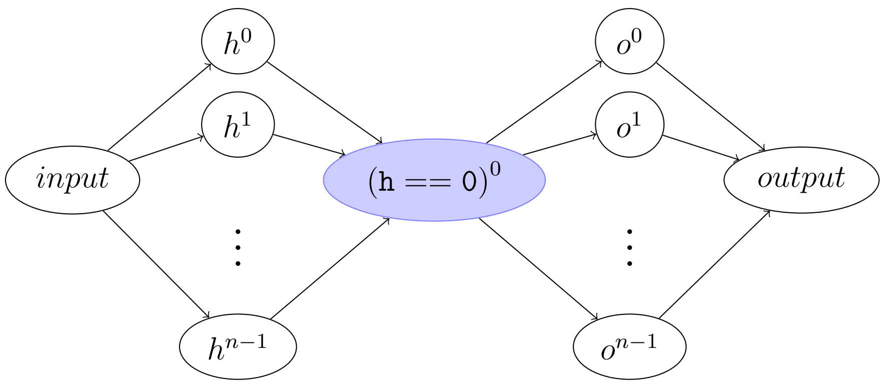


Figure 4.3.: The leakage graph for Listing 4.4

The analysis of this program works as before and computes the following output value:

$$o = \Big(\underbrace{(\hat{u},\ \{(\texttt{h} == \texttt{0})^0\})}_{o^{n-1}}, \ldots, \underbrace{(\hat{u},\ \{(\texttt{h} == \texttt{0})^0\})}_{o^0} \Big)$$

$(\texttt{h} == \texttt{0})^0 = (\hat{u},\ \{h^1, h^0\})$ is the least significant bit of the value of the node that the ϕ-node control-depends on. The number of high dependent unknown output bits and the number of input bits is n. Therefore, the basic leakage approximations calculate a leakage of $C(o) = min(n, n) = n$. It is clear that the example program has only two different outputs and therefore a leakage of 1. The generalised approach approximates the leakage to be 1. The bit dependency graph is given in Figure 4.3.

4.2.4. Leakage computation for arbitrary security lattices

We can extended the leakage computation to calculate the leakage for an attacker on security level α. The leakage computation works mainly as before, but uses all input bits with a security level $\not\leq \alpha$ as secret input and all bits of outputs with a security level $\leq \alpha$ as public output.

5. Extension: Increasing Precision

This chapter explains how the basic analysis, described in Chapter 3, can be extended to increase its precision. This finalises the analysis for loop-free programs. It is followed by a specification of all directly supported operators.

5.1. Extended Analysis

The analysis described in Chapter 3 does not propagate information on bits that can be easily obtained by analysing evaluated conditions that an expression control depends on. If we know for example that the condition `x == 1` evaluated to *true* in the path to the current statement, then we also know that `x` has the value `1` there. Such information can be used to increase the precision of the analysis. The described propagation is based on the notion of propagated predicates, first formalised by Wegbreit [30]. This section starts with an introduction of the basic propagation idea, followed by a formal description and the specification of the equality operator.

5.1.1. Basic idea and formalisation

The basic idea is to have a local mapping called *mods* to store the gathered information.

Definition 5.1 (*mods* mapping)**.**

$$mods = \underbrace{Identity}_{\text{bit}} \rightharpoonup \underbrace{Identity}_{\text{replacement}}$$

Each *mods* mapping maps a bit to a replacement bit in a specific context, representing the gathered knowledge on the relation between the two bits. We define the supremum relation for such mappings as:

$$\begin{aligned} mods \sqcup mods &\to mods \\ a \sqcup b &\mapsto \lambda x. \begin{cases} a(x) & a(x) = b(x) \lor b(x) = None \\ b(x) & a(x) = None \\ None & \text{else} \end{cases} \end{aligned}$$

The *mods* mappings are created whenever a condition of a conditional statement like an if-statement is evaluated. Such newly gathered *mods* mappings are stored in the *modsMap (Definition 5.2)* for each branch of the conditional statement.

Example The gathered knowledge for a basic case can be seen in the following example with one bit values, given in Listing 5.1.

```
1 if (h){ // the then-branch is only evaluated if h is 1
2   o1 = h
3   ...
4 }
5 ...
```

Listing 5.1: Basic example with boolean values

The analysis can statically assert that `h` has the value `1` in the then-branch of the if-statement. During the evaluation of this branch, the *mods* mapping for this context should therefore contain a mapping from h^0 to $(\hat{1}, \varnothing)$. This knowledge is stored in the *modsMap*. When `h` is then accessed in line 2, h^0 should be replaced by $(\hat{1}, \varnothing)$. Without taking the condition into account, `o1` would have the value $\big((\hat{u},\ \{h^0\})\big)$.

Definition 5.2 (*modsMap* mapping)**.** The *modsMap* mapping maps each branch of every conditional statement to a *mods* mapping:

$$\begin{aligned} \mathit{modsMap} &: \mathit{Branch} \to \mathit{mods} \\ \mathit{Branch} &= \mathit{Node}_{\mathit{condition}} \times \underbrace{\mathbb{B}}_{\{\mathit{true},\mathit{false}\}} \end{aligned}$$

The *modsMap* is updated after the evaluation of the condition of every conditional statement, to contain the newly gathered knowledge for each branch of the conditional statement.

If the current conditional statement is part of the branch of another (outer) conditional statement, then the *mods* of this outer branch also apply. The *mods* of inner branches are prioritised over *mods* of the outer branches.

Example In the prior example, the analysis would add the following to the *modsMap* after analysing the condition `h`:

$$\begin{aligned} (\mathtt{h}, \mathit{true}) &\mapsto \big(h^0 \mapsto (\hat{1}, \varnothing)\big) \\ (\mathtt{h}, \mathit{false}) &\mapsto \big(h^0 \mapsto (\hat{0}, \varnothing)\big) \end{aligned}$$

The evaluation of the expression `h` in line 2 belongs to the then-branch, therefore the *mods* of *modsMap*(`h`, *true*) are applied. If the if-statement is part of another branch, then the *mods* of this branch are applied the same way.

Before we describe how to acquire such knowledge from more complex conditional expressions, it is important to know how the *mods* mapping is used when accessing a bit.

Definition 5.3 (*replace* function)**.** Every time a bit is accessed in the *evaluate* function *(Definition 3.10)*, the modifications stored in the *modsMap* mapping are applied using the *replace* function.

$$replace : \underbrace{Node}_{\text{current PDG node}} \times \underbrace{Identity}_{\text{passed bit}} \to \underbrace{Identity}_{\text{replacement}}$$

It replaces the passed bit using the first applicable replacement from the current branch upwards that is stored in the *modsMap*.

This process is repeated with the replacement, as long as there is a replacement available. In the example from Listing 5.1, this results in a call of $replace(\texttt{o1}, h^0)$ that returns $(\hat{1}, \varnothing)$.

The version of *replace* that accepts whole bit-string values as its argument applies the *replace* function on every bit with the same node as the first argument.

With this, the *evaluate* function can be altered to call the *replace* function properly. The altered pseudo-code of the *evaluate* function is given in Listing 5.2.

```
def evaluate(node: Node):
  args = [replace(node, nodeValue(p)) for p in paramNode(node)]
  for i, new in enumerate(op(node)(args)):
    bitMap(node^i) = bitMap(new)
```

Listing 5.2: Altered *evaluate* function that uses the *replace* function

Example for updating *modsMap* The example presented in Listing 5.3 explains the proper updating of the *modsMap*. This program uses two bit values. In the case of if-statements we know that the least significant bit of the conditional expression has to be $1 = (\hat{1}, \varnothing)$ in the then-branch and 0 in the else-branch. Therefore the evaluation of the condition in line 1 adds the following mappings to *modsMap*:

$$(\texttt{h}, true) \mapsto \left(h^0 \mapsto (\hat{1}, \varnothing)\right)$$
$$(\texttt{h}, false) \mapsto \left(h^0 \mapsto (\hat{0}, \varnothing)\right)$$

With the current evaluation technique, no *mods* would be applied to the arguments of the ϕ expression. This is because the ϕ-operator is a special operator as it takes operands originating from different branches.

Alteration of *evaluate* **for** ϕ**-expressions** During a real execution, only one operand is chosen. Therefore the *mods* applied to each operand have to be related to the branch that caused the operand to be passed to the operator. Such a branch is acquired via the function

$$paramBranch : Node \times \mathbb{N}_0 \to Branch$$

```
if (h) {
  o1 = h
} else {
  o2 = 0b11
}
o = ϕ(o1, o2)
```

Listing 5.3: Example for a program in which the replacements that are applied to the values should depend on the branch that they originate from

returning the branch for each operand, referenced by its index. With this the *evaluate* function is modified as given in Listing 5.4.

```
def evaluate(node: Node):
  args = [replace(paramBranch(node, i), nodeValue(p))
          for i, p in enumerate(paramNode(node))]
  for i, new in enumerate(op(node)(args)):
    bitMap(node^i) = bitMap(new)
```

Listing 5.4: Altered *evaluate* function that takes the parameter branch into account

This is equivalent to the previous version for all but the ϕ-nodes, as the following condition holds for all other nodes μ: $\forall i : \mathit{paramBranch}(\mu, i) = \mathit{branch}(\mu)$.

Definition 5.4 (*repl* mapping)**.** The operators store a function in the *repl* mapping for every bit. This function returns the *mods* mapping for an asserted value of the bit that the function is associated with.

$$\mathit{repl} : \underbrace{\mathit{Identity}}_{\text{associated bit}} \to (\underbrace{\mathit{Identity}}_{\text{asserted value}} \to \underbrace{\mathit{mods}}_{\text{resulting knowledge on bits}})$$
$$b \mapsto \lambda a.(b \mapsto a)$$

The supremum on *repl* entries is defined point-wise.

This functions stored in this mapping are used to populate the *modsMap* mapping after the evaluation of a condition, e.g. after analysing a condition μ the *modsMap* is updated as follows:

$$\mathit{modsMap}\big((\mu, \mathit{true})\big) = \mathit{repl}(\mu^0)\big((\hat{1}, \varnothing)\big)$$
$$\mathit{modsMap}\big((\mu, \mathit{false})\big) = \mathit{repl}(\mu^0)\big((\hat{0}, \varnothing)\big)$$

The gathering of knowledge from conditions is generalised in the following using the *repl* mapping.

```
if (e1 == e2) {
  ...
} else {
  ...
}
```

Listing 5.5: Basic example with the equality operator

```
if (h == 0) {
  o1 = h
} else {
  o2 = 0
}
o = ϕ(o1, o2)
```

Listing 5.6: Basic if-statement example

5.1.2. Equality operator

Consider the following example presented in Listing 5.5. In the then-branch, the condition `e1 == e2` is true and therefore the results of the evaluation of `e1` and `e2` have to be equal, in particular their bit values. So if we know that the ith bit of `e1` evaluates to 1, then we also know that the ith bit of `e2` evaluates to 1 too, even if $e2^i$ was statically unknown before. The same applies to 0. If both bits are statically known, then we cannot gain more knowledge about them. In general, we cannot state anything about the else-branch. The resulting addition to *repl* can be formalised as:

$$\begin{aligned} repl\big((\texttt{e1} == \texttt{e2})^0\big) &= \lambda a.\big((\texttt{e1} == \texttt{e2})^0 \mapsto a\big) \sqcup \\ &\bigsqcup_{i=0}^{n-1} \begin{cases} repl(e1^i)(e2^i) & v_a = \hat{1} \wedge v_{e1}^i = \hat{u} \wedge v_{e2}^i \in \{\hat{0}, \hat{1}\} \\ repl(e2^i)(e1^i) & v_a = \hat{1} \wedge v_{e2}^i = \hat{u} \wedge v_{e1}^i \in \{\hat{0}, \hat{1}\} \\ () & \text{else} \end{cases} \end{aligned}$$

The *repl* instantiation for the expression result uses the *repl* instantiations of its operands recursively. This formalisation of the operator is used in the following example.

Example with an equality operator The code of this example is given in Listing 5.6.

After evaluating the conditional expression, the following is added to *repl*, inlining the basic *mods*:

$$\begin{aligned} \mathit{repl}\big((\texttt{h} == \texttt{0})^0\big) \;&=\; \lambda a.\big((\texttt{h} == \texttt{0})^0 \mapsto a\big)\sqcup \\ &\qquad \bigsqcup_{i=0}^{n-1} \begin{cases} 0^i \mapsto h^i & v_a = \hat{1} \wedge v_h^i = \hat{u} \wedge v_0^i \in \{\hat{0}, \hat{1}\} \\ h^i \mapsto 0^i & v_a = \hat{1} \wedge v_0^i = \hat{u} \wedge v_h^i \in \{\hat{0}, \hat{1}\} \\ () & \text{else} \end{cases} \end{aligned}$$

Then in the then-branch the *mods* apply that are generated by $\mathit{repl}\big((\texttt{h} == \texttt{0})^0\big)\big((1, \varnothing)\big)$, this results in the mappings:

$$\begin{aligned} (\texttt{h} == \texttt{0})^0 &\mapsto a \\ h^i &\mapsto (\hat{0}, \varnothing) \qquad 0 \le i < n \end{aligned}$$

As a result, the expression ϕ`(o1, o2)` in line 5 evaluates to a constant value, resulting in no leakage of secret information during any execution of the program.

A key observation is that we can exchange the i$^{\text{th}}$ bit in `e1` with the i$^{\text{th}}$ bit in `e2` and vice versa if `e1` is known to be equal to `e2`, and the analysis is still correct.

The following deals with both bits being statically unknown: The aim is to choose the bit that results in a lower leakage and therefore a more precise leakage calculation.

Definition 5.5 (*choose* heuristic)**.** A simple heuristic is to choose the bit that has the lower intrinsic leakage $C\big((\cdot)\big)$.

$$\begin{aligned} \mathit{choose} : \mathit{Identity} \times \mathit{Identity} &\to \mathit{Identity} \\ (a, b) &\mapsto \begin{cases} a & C\big((a)\big) \le C\big((b)\big) \\ b & \text{else} \end{cases} \end{aligned}$$

The bit that is not chosen is returned by the function *notChosen*:

$$\begin{aligned} \mathit{notChosen} : \mathit{Identity} \times \mathit{Identity} &\to \mathit{Identity} \\ (a, b) &\mapsto \begin{cases} a & \mathit{choose}(a, b) = b \\ b & \text{else} \end{cases} \end{aligned}$$

This heuristic does not result in optimal results, as it cannot take the whole bit dependency graph into account. A solution to this problem is presented in Appendix A.

Usage of the heuristic We can extend the default value of the *repl* mapping to call the *choose* function:

$$\mathit{repl}(b) = \lambda a.\big(b \mapsto \mathit{choose}(a, b)\big)$$

```
if (h == l) {
  o1 = h
} else {
  o2 = 0b11
}
o = ϕ(o1, o2)
```

Listing 5.7: Example in which the *choose* heuristic increases the precision of the analysis

Another usage is the simplification of the *repl* instantiation for the equality operator:

$$repl\big((\mathtt{e1} == \mathtt{e2})^0\big) \;=\; \lambda a.\Big(\big(\mathtt{e1} == \mathtt{e2}\big)^0 \mapsto choose\big(a, (\mathtt{e1} == \mathtt{e2})^0\big)\Big) \sqcup \bigsqcup_{i=0}^{n-1} \begin{cases} repl\big(notChosen(e1^i, e2^i)\big)\big(choose(e1^i, e2^i)\big) & v_a = \hat{1} \\ () & \text{else} \end{cases}$$

A complete specification of the equality operator and other operators can be found in Section 5.2.

There are many cases where the *choose* heuristic increases the precision of the analysis. One such case is given in the following example whose code is presented in Listing 5.7

The analysis of the conditional expression results in the addition of the following mapping to *repl*:

$$repl\big((\mathtt{h} == \mathtt{l})^0\big) \;=\; \lambda a.\Big((\mathtt{h} == \mathtt{l})^0 \mapsto choose\big(a, (\mathtt{h} == \mathtt{l})^0\big)\Big) \sqcup \bigsqcup_{i=0}^{1} repl\big(choose(h^i, l^i)\big)\big(notChosen(h^i, l^i)\big)$$

For the then-branch the *mods* are gathered by calling $repl\big((\mathtt{h} == \mathtt{l})^0\big)\big((\hat{1}, \varnothing)\big)$ which evaluates to:

$$\begin{gathered} (\mathtt{h} == \mathtt{l})^0 \mapsto (\hat{1}, \varnothing) \\ h^1 \mapsto l^1 \\ h^0 \mapsto l^0 \end{gathered}$$

The *mods* for the else-branch are gathered similarly:

$$(\mathtt{h} == \mathtt{l})^0 \mapsto (\hat{0}, \varnothing).$$

Using these *mods*, the expression ϕ`(o1, o2)` evaluates to

$$\big((\hat{u},\ \{l^1, (\mathtt{h} == \mathtt{l})^0\}),\ (\hat{u},\ \{l^0, (\mathtt{h} == \mathtt{l})^0\})\big).$$

Therefore the approximated leakage of the program is 1, using the generalised leakage approximation.

```
1  if (h1) {
2    x1 = h1
3  } else {
4    x2 = 0b11
5  }
6  x = ϕ(x1, x2) // $= \big((\hat{u},\ \{h1^0, h1^0\}),\ (\hat{1}, \varnothing)\big)$
7  if (h2) {
8    y1 = h2
9  } else {
10   y2 = 0b11
11 }
12 y = ϕ(y1, y2) // $= \big((\hat{u},\ \{h2^1, h2^0\}),\ (\hat{1}, \varnothing)\big)$
13 s = x == y
14 if (s) {
15   o1 = y
16 } else {
17   o2 = 0b11
18 }
19 o = ϕ(o1, o2)
20 p = h1
```

Listing 5.8: Complex example that results in non-trivial cuts for the minimum-vertex-cut based leakage approximation

5.1.3. Complex example

The following is a more complex example that shows how the analysis works and that the minimum-vertex-cut based leakage approximation can lead to non-trivial cuts. The example, see Listing 5.8, uses two bit values, two high input variables `h1` and `h2`, and three output variables `o`, `p` and `s`:

The analysis of the code until the line 12 is analogous to the prior examples and therefore omitted here. Evaluating line 13 results in the addition of the following to *repl*:

$$repl(\underbrace{s^0}_{(\mathtt{x==y})^0}) = \lambda a.(s \mapsto a) \sqcup \bigsqcup_{i=0}^{1} repl\big(notChosen(x^i, y^i)\big)\big(choose(x^i, y^i)\big)$$

s^0 evaluates to $(\hat{u},\ \{x^1, y^1\})$. This results in the following bit modifications for the then-branch by calling $repl(s^0)\big((\hat{1}, \varnothing)\big)$:

$$\begin{aligned} s &\mapsto (\hat{1}, \varnothing) \\ y^1 &\mapsto x^1 \\ y^0 &\mapsto x^0, \end{aligned}$$

resulting in y being replaced by x in the expression ϕ`(o1, o2)` in line 19. Therefore, the ϕ-expression evaluates to $\big((\hat{u},\ \{x^1, s^0\}),\ \ (\hat{1}, \varnothing)\big)$. This results in a leakage of 3. The minimum-vertex-cut approach gives us the following bit dependency graph presented in Figure 5.1.

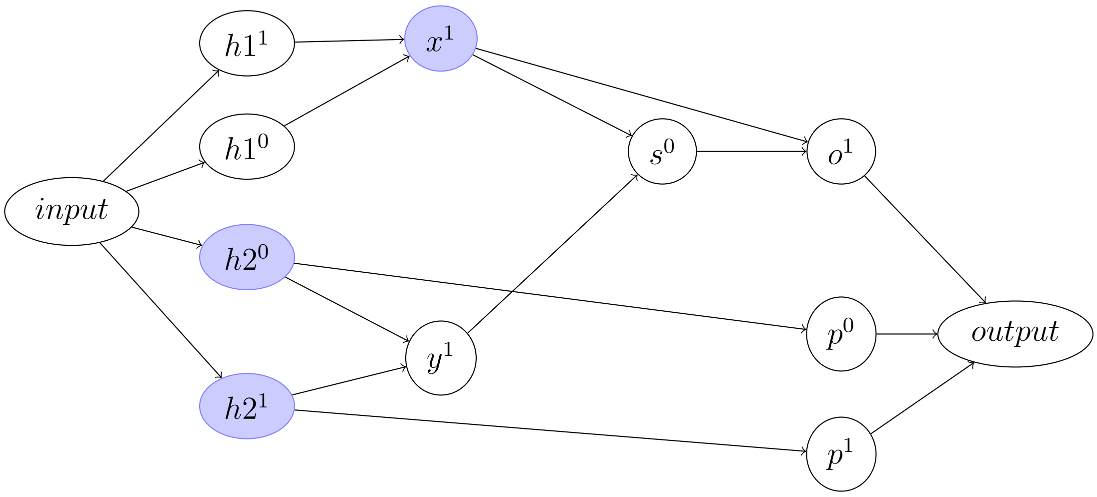


Figure 5.1.: Bit dependency graph for the example program given in Listing 5.8

5.2. Operator Specification

This section extends the list of operators from Section 3.3 and includes the *repl* instantiations for each operator. Only few operators are specified, other operators can be transformed into expressions consisting of these basic operators before the analysis.

In the following, x is the first operand, y the second operand for binary operators and r the result of each operator. The replacement function is

$$v_r^i \in \{\hat{0}, \hat{1}\} \Rightarrow repl(r^i) = \lambda a.()$$

for all constant return bits r^i. This is omitted in the operator specifications for brevity. The specifications only include the replacement functions for non-constant bits. Another simplification is that assigning an argument bit α^j directly to a result bit r^i implicitly creates a new bit that copies the *repl* entry and solely depends on the bit α^j:

$$\alpha \in \{x, y\} : r^i = \alpha^j$$
$$\equiv v_r^i = v_\alpha^j \wedge d_r^i = \begin{cases} \{\alpha^j\} & v_\alpha^j \in \{\hat{0}, \hat{1}\} \\ \varnothing & \text{else} \end{cases} \wedge repl(r^i) = repl_{[\alpha^j \to r^i]}(\alpha^j)$$

5.2.1. Bitwise operators

Bitwise operators calculate each result bit using the corresponding operand bits of the same index. The following specification of four operators assumes $0 \leq i < n$.

Bitwise negation-operator

The negation-operator negates every bit of its operand.

$$\begin{aligned}
v_r^i &= \begin{cases} \hat{1} & v_x^i = \hat{0} \\ \hat{0} & v_x^i = \hat{1} \\ \hat{u} & \text{else} \end{cases} \\
d_r^i &= \begin{cases} \{d_x^i\} & v_x^i = \hat{u} \\ \varnothing & \text{else} \end{cases} \\
\mathit{repl}(r^i) &= \lambda a.\big(r^i \mapsto \mathit{choose}(a, r^i)\big) \\
&\quad \sqcup \begin{cases} \mathit{repl}(x^i)\big((\hat{0}, d_a)\big) & v_a = \hat{1} \\ \mathit{repl}(x^i)\big((\hat{1}, d_a)\big) & v_a = \hat{0} \\ \mathit{repl}(x^i)(a) & v_a = \hat{u} \\ () & \text{else} \end{cases}
\end{aligned}$$

Bitwise or-operator

A detailed explanation of the operator is given in Section 3.3, the only addition to the previous specification is the alteration of *repl*. The *repl* instantiation depends on whether the value of one operand's bit determines the result alone. If this is the case, the *repl* instantiation of this bit is used. If the result of the or-expression has value $\hat{0}$, it can be asserted that both operand bits are $\hat{0}$, too.

$$\begin{aligned}
\mathit{repl}(r^i) &= \lambda a.\big(r^i \mapsto \mathit{choose}(a, r^i)\big) \\
&\quad \sqcup \begin{cases} \mathit{repl}(x^i)(a) & v_a = \hat{1} \wedge v_{y^i} = \hat{0} \\ \mathit{repl}(y^i)(a) & v_a = \hat{1} \wedge v_{x^i} = \hat{0} \\ \mathit{repl}(x^i)(a) \sqcup \mathit{repl}(y^i)(a) & v_a = \hat{0} \\ () & \text{else} \end{cases}
\end{aligned}$$

The bitwise and-operator can be defined analogously or by using the equivalence of $a \wedge b$ and $\neg(\neg a \vee \neg b)$.

Bitwise exclusive-or-operator

The exclusive-or-operator returns 1 only if both operand bits have different values at runtime. It can be seen as a bitwise not-equal operator.

$$v_r^i = \begin{cases} \hat{1} & \{\hat{0},\hat{1}\} = \{v_y^i, v_y^i\} \\ \hat{0} & v_x^i = v_y^i = \alpha, \alpha \in \{\hat{0},\hat{1}\} \\ \hat{u} & \text{else} \end{cases}$$

If the result is statically unknown, then it depends on the statically unknown operand bits.

$$d_r^i = \begin{cases} \{\alpha^i | \alpha \in \{x,y\}, v_\alpha^i = \hat{u}\} & v_r^i = \hat{u} \\ \varnothing & \text{else} \end{cases}$$

The following is split into three parts, depending on the bit value of the result. The notation $\bigsqcup_{\{\alpha,\beta\}=\{x,y\}}$ expresses the combination off the cases $\alpha = x, \beta = y$ and $\alpha = y, \beta = x$.

$$repl(r^i) = \lambda a.(r^i \mapsto choose(a, r^i)) \sqcup$$

$$\bigsqcup_{\{\alpha,\beta\}=\{x,y\}} \begin{cases} repl(\alpha^i)((\hat{1}, d_a)) & v_a = \hat{1} \wedge v_{\beta^i} = \hat{0} \\ repl(\alpha^i)((\hat{0}, d_a)) & v_a = \hat{1} \wedge v_{\beta^i} = \hat{1} \\ repl(\alpha^i)((\hat{u}, d_{\beta^i} \cup d_a)) & v_a = \hat{1} \wedge v_{\alpha^i} = \hat{u} \wedge v_{\beta^i} = \hat{u} \wedge \beta^i = choose(\alpha^i, \beta^i) \\ repl(\alpha^i)((\hat{0}, d_a)) & v_a = \hat{0} \wedge v_{\beta^i} = \hat{0} \\ repl(\alpha^i)((\hat{1}, d_a)) & v_a = \hat{0} \wedge v_{\beta^i} = \hat{1} \\ repl(\alpha^i)((\hat{u}, d_{\beta^i} \cup d_a)) & v_a = \hat{0} \wedge v_{\alpha^i} = \hat{u} \wedge v_{\beta^i} = \hat{u} \wedge \beta^i = choose(\alpha^i, \beta^i) \\ () & \text{else} \end{cases}$$

ϕ-operator

A detailed explanation of this operator is given in Section 3.3. The ϕ-operator works with an arbitrary number of operand values α_j. The only addition to the previous explanation is the alteration of *repl* for the case that all but one operand's bit is statically known, else no knowledge can be propagated.

$$repl(r^i) = \lambda a.(r^i \mapsto choose(a, r^i))$$

$$\sqcup \begin{cases} repl(\alpha_k^i)(a) & |\{\alpha_j | v_{\alpha_j}^i = \hat{u}\}| = 1 \wedge v_{\alpha_k}^i = \hat{u} \\ () & \text{else} \end{cases}$$

5.2.2. Comparison operators

All comparison operators return a boolean:

$$r^i = (\hat{0}, \varnothing) \qquad 0 < i < n$$

Only two operators are specified in the following, all other comparison operators can be replaced prior to the analysis with a combination of these.

Equality operator

A detailed explanation of the operator is given in Section 3.3, the only addition is the alteration of *repl* explained in Section 5.1.2:

$$\begin{aligned}
\mathit{repl}(r^1) = \lambda a.\big(r^1 \mapsto \mathit{choose}(a, r^1)\big) \sqcup \\
\bigsqcup_{i=1}^{n} \begin{cases} \mathit{repl}\big(\mathit{notChosen}(x^i, y^i)\big)\big(\mathit{choose}(x^i, y^i)\big) & v_a = \hat{1} \\ () & \text{else} \end{cases}
\end{aligned}$$

The inequality operator can be defined accordingly.

Less operator

The less operator specified in the following has to explicitly deal with the sign bit, i.e. the highest significant bit.

Before defining the operator, two helper functions are defined:

The *diff*-function returns the maximum index at which both operands have the same constant bit value for all lower bits.

$$\begin{aligned}
\mathit{diff} : \mathit{Value} \times \mathit{Value} \rightharpoonup \mathbb{N}_0 \\
(\alpha, \beta) \mapsto \max\{j | 0 \leq j < n-1 \wedge v_\alpha^j = v_\beta^j \wedge v_\alpha^j \in \{\hat{0}, \hat{1}\}\}
\end{aligned}$$

There are two basic cases for calculating the bit value of the result bit: If the first operand is negative and the second operand is positive, then the result is $\hat{1}$. Conversely, the result is $\hat{0}$ for the opposite case.

The two remaining cases of both operands being positive or negative are more complicated. If both operands are positive and they differ at index $\mathit{diff}(x, y)$ with constant bits, then the result is $\hat{1}$ if the bit value of y at this index is $\hat{1}$ and $\hat{0}$ otherwise. If both operands are negative and they differ at index $\mathit{diff}(x, y)$ with constant bits, then the result is $\hat{1}$ if the bit value of x at this index is $\hat{1}$ and $\hat{0}$ otherwise.

$$v_r^0 = \begin{cases}
\hat{1} & v_x^{n-1} = \hat{1} \wedge v_x^{n-1}, v_y^{n-1} \in \{\hat{0}, \hat{1}\} \wedge v_x^{n-1} \neq v_y^{n-1} \\
\hat{0} & v_x^{n-1} = \hat{0} \wedge v_x^{n-1}, v_y^{n-1} \in \{\hat{0}, \hat{1}\} \wedge v_x^{n-1} \neq v_y^{n-1} \\
v_y^{\mathit{diff}(x,y)} & v_x^{n-1} = \hat{0} \wedge v_y^{n-1} = \hat{0} \wedge \mathit{diff}(x, y) \neq \mathit{None} \wedge v_x^{\mathit{diff}(x,y)}, v_y^{\mathit{diff}(x,y)} \in \{\hat{0}, \hat{1}\} \\
v_x^{\mathit{diff}(x,y)} & v_x^{n-1} = \hat{1} \wedge v_y^{n-1} = \hat{1} \wedge \mathit{diff}(x, y) \neq \mathit{None} \wedge v_x^{\mathit{diff}(x,y)}, v_y^{\mathit{diff}(x,y)} \in \{\hat{0}, \hat{1}\} \\
\hat{u} & \text{else}
\end{cases}$$

The result bit depends on all unknown bits of the operands.

$$d_r^0 = \begin{cases} \{\alpha^i | \alpha \in \{x, y\}, v_\alpha^i = \hat{u}\} & v_r^i = \hat{u} \\ \varnothing & \text{else} \end{cases}$$

Good heuristics for generating *mods* implicated by a less-expression are difficult to construct, they are outside the scope of this work.

$$repl(r^0)(a) = r^0 \mapsto choose(a, r^0)$$

5.2.3. Single bit operators

The following two operators are useful for working directly on bits in the program.

Bit selection operator

The bit selection operator, `x[j]`, returns the jth bit of x. The resulting value consists of zeros, besides the least significant bit which is set to the jth bit of x. j has to be a valid bit index that is statically known.

$$\begin{aligned} r^i &= (\hat{0}, \varnothing) \qquad 0 < i < n \\ r^0 &= (v_x^j, d_x^j) \\ repl(r^0) &= \lambda a.\big(r^0 \mapsto choose(a, r^0)\big) \sqcup repl(x^j)(a) \end{aligned}$$

Bit place operator

The bit place operator `[j]x` returns a value that consists of zeros, besides the jth bit which is replaced by the least significant bit of x. j has to be a valid bit index that is statically known.

$$\begin{aligned} r^i &= (\hat{0}, \varnothing) \qquad 0 \leq i < n \land i \neq j \\ r^j &= (v_x^0, d_x^0) \\ repl(r^j) &= \lambda a.\big(r^j \mapsto choose(a, r^j)\big) \sqcup repl(x^0)(a) \end{aligned}$$

5.2.4. Shift operators

The following presents the specification of the two logical shift operators.

Left shift operator

The left shift operator shifts the bits y to the left and inserts zeros as needed with $0 \leq i < n$:

$$\text{iff } y \text{ is a constant integer} \geq 0$$
$$r^i = \begin{cases} x^{i-y} & i - y \geq 0 \\ (0, \varnothing) & \text{else} \end{cases}$$

$$\text{iff } y \text{ is a constant integer} < 0$$
$$r^i = \begin{cases} x^{i-y} & i - y < n \\ (0, \varnothing) & \text{else} \end{cases}$$

$$\text{else}$$
$$r^i = (\hat{u}, x \cup y)$$

Right shift operator

The right shift operator shifts the bits y to the right and inserts zeros as needed with $0 \leq i < n$:

$$\text{iff } y \text{ is a constant integer} \geq 0$$
$$r^i = \begin{cases} 0 & i + y > n - 1 \\ x^{i+y} & \text{else} \end{cases}$$

$$\text{iff } y \text{ is a constant integer} < 0$$
$$r^i = \begin{cases} 0 & i + y < 0 \\ x^{i+y} & \text{else} \end{cases}$$

$$\text{else}$$
$$r^i = (\hat{u}, x \cup y)$$

5.2.5. Arithmetic operators

Bit lattices are not well suited for modelling arithmetic operators [31]. The following specifications are sound approximations of four operators. The most precise specification would result in transforming each arithmetic operator into an expression using only bitwise operators before the analysis. The addition operator is the only operator without an imprecise version since the created expressions are comparably small.

Multiplication operator

This imprecise specification handles powers of two precisely.

$$r^{n-1} = \begin{cases} (\hat{0}, \varnothing) & v_x^{n-1} = v_y^{n-1} \wedge v_x^{n-1} \in \{\hat{0}, \hat{1}\} \\ (\hat{1}, \varnothing) & v_x^{n-1} \neq v_y^{n-1} \wedge v_x^{n-1}, v_y^{n-1} \in \{\hat{0}, \hat{1}\} \\ (\hat{u}, x \cup y) & \text{else} \end{cases}$$

iff x and y are constant values

$$r^{[0,n-2]} = \mathit{valueOf}(x \cdot y)$$

else if $|y|$ is a constant power of two

$$\forall i \in [0, n-2] : r^i = \begin{cases} x^{i-|y|} & i - |y| \geq 0 \\ (0, \varnothing) & \text{else} \end{cases}$$

else if $|x|$ is a constant power of two

$$\forall i \in [0, n-2] : r^i = \begin{cases} y^{i-|x|} & i - |x| \geq 0 \\ (0, \varnothing) & \text{else} \end{cases}$$

else

$$\forall i \in [0, n-2] : r^i = (\hat{u}, x \cup y)$$

An implementation based on the transformation into basic operators can be done based on for example the circuit proposed by Dadda [32].

Division operator

An imprecise version that assumes that a division by zero cannot occur, as exceptions are not supported by the analysis.

$$r^{n} = \begin{cases} (\hat{0}, \varnothing) & v_x^{n-1} = v_y^{n-1} \wedge v_x^{n-1} \in \{\hat{0}, \hat{1}\} \\ (\hat{1}, \varnothing) & v_x^{n-1} \neq v_y^{n-1} \wedge v_x^{n-1}, v_y^{n-1} \in \{\hat{0}, \hat{1}\} \\ (\hat{u}, x \cup y) & \text{else} \end{cases}$$

iff x and y are constant values

$$r^{[0,n-2]} = valueOf(x/y)$$

else if $|y|$ is a constant power of two

$$\forall i \in [0, n-2] : r^i = \begin{cases} 0 & i - |y| < n \\ x^{i-|y|} & \text{else} \end{cases}$$

else

$$\forall i \in [0, n-2] : r^i = (\hat{u}, x \cup y)$$

```
def halfadder(a, b):
  return (a ^ b, a & b)

def fulladder(a, b, c):
  (rt, c1) = halfadder(a, b)
  (r, c2)  = halfadder(rt, c)
  r2 = c1 | c2
  return (r, r2)

def adder(a, b):
  result = 0
  carry = 0
  for i in range(0, n):
    (r, carry) = fulladder(a[i], b[i], carry)
    result = result | [i]r
  return result
```

Listing 5.9: Pseudo-code for the addition operator

Modulo operator

The modulo operator treats the case of a constant divisor that is a power of two specially:

$$r^n = \begin{cases} \hat{0} & y \text{ is a constant power of two and } v_x^{n-1} = \hat{0} \\ \hat{1} & y \text{ is a constant power of two, } x \text{ is contant and } x \bmod y < 0 \\ \hat{u} & \text{else} \end{cases}$$

if $y = 2^\alpha$ for an $\alpha \in \mathbb{N}_0$

$$\begin{aligned} r^i &= x^i & 0 \le i \le \alpha \\ r^j &= (\hat{0}, \varnothing) & \alpha < j < n \end{aligned}$$

Addition operator

The transformation for the addition operator is given as pseudo-code in Listing 5.9. It results in a half-adder based construct that should be inlined before the actual analysis. This results in $9n$ operations per addition.

The addition operator can also be implemented by using the algorithm given in Listing 5.9 to compute the result of each addition operation and then altering the result bits so that they only depend on the operand bits that they originally transitively depended on.

6. Extension: Loop Analysis

The previous chapters elaborated on the analysis of loop-free PDGs. This chapter introduces a version of the analysis that allows to evaluate PDGs with arbitrary reducible control flows. A control flow graph is reducible when the iterative replacement of the smallest strongly connected sub-graph with a single node results in a linear graph [33]. Programs with reducible control graphs closely match the notion of control flow graphs created by control-flow constructs of the Java language [34, p. 377].

6.1. Evaluation Function Alteration

The idea behind the alteration of the *evaluate* function is to merge the evaluation result with the old bits for a PDG node stored in *bitMap* whenever a PDG node is reevaluated. This merge is the bitwise supremum. This alteration of the evaluation function from Section 5.1 results in the version that is presented in Listing 6.1.

A problem that arises from the generalised leakage approximations is that a condition bit that belongs to the evaluation of a loop can leak more than one bit, as the loop condition can be executed multiple times with changing argument values.

Definition 6.1 (Weight)**.** All condition bits that are computed in a condition as a part of a loop are assigned an infinite weight to soundly approximate loops. Every operator assigns this weight. A condition bit with an infinite weight can leak all bits that it depends on. Therefore, such bits are excluded from the minimal-set of fixed bits in Section 4.2. This is realised in the minimum-vertex-cut calculation by

```
def evaluate(node: Node):
  args = [replace(paramBranch(node, i), nodeValue(p))
          for i, p in enumerate(paramNode(node))]
  new_value = op(node)(args)
  for (old, new) in zip(nodeValue(node), new_value):
    repl(old)    = repl(old) ⊔ repl(new)
    bitMap(old) = bitMap(old) ⊔ bitMap(new)
```

Listing 6.1: Altered version of the *evaluate* function that supports the reevaluation of nodes

```
while (h != o) {
  o = o + 1
}
```

```
int o1 = 0
while [o2 = ϕ(o1, o3)] (h != o2) {
  o3 = o2 + 1
}
o = o2
```

Listing 6.2: Counting loop example. On the right hand side shows the program in SSA-form. The statement `o2 = ϕ(o1, o3)` is executed before all other statements and expressions of the loop.

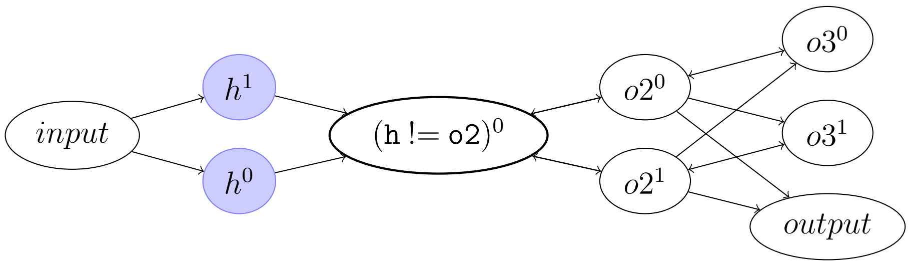


Figure 6.1.: Dependency graph for the program given in Listing 6.2. Note that "↔" represents edges in both directions. The node with the thick border has infinite weight.

assigning a weight to each bit-vertex.

$$weight(\mathbf{r^0}) = \begin{cases} \infty & v_r^0 = \hat{u} \\ & \wedge \text{ node is part of a condition} \\ & \wedge \text{ is inside a loop or a loop condition} \\ 1 & \text{else} \end{cases}$$

Termination The fix point iteration using the *evaluate* function terminates, as the bit lattice has a fixed height and the transfer function is monotone.

6.2. Example

This examples shows the importance of setting the infinite weight for loop conditions. The following describes the analysis of the program given in Listing 6.2 for two bit values. The modification of *repl* is omitted for brevity. The analysis steps are given in the form of changing bits and *weight* entries.

First evaluation of the loop:

$$\begin{aligned}
nodeValue(\underbrace{\phi(o1, o3)}_{o2}) &= \big((\hat{0}, \varnothing), & (\hat{0}, \varnothing)\big)\\
nodeValue(\underbrace{\texttt{h != o2}}_{\text{condition}}) &= \big((\hat{0}, \varnothing), & (\hat{u},\ \{h^1, h^0\})\big)\\
weight\big((\texttt{h != o2})^0\big) &= \infty\\
nodeValue(\underbrace{o2 + 1}_{o3}) &= \big((\hat{0}, \varnothing), & (\hat{1}, \varnothing)\big)
\end{aligned}$$

Next iteration of the loop, as the bits changed:

$$\begin{aligned}
nodeValue(\underbrace{\phi(o1, o3)}_{o2}) &= \big((\hat{0}, \varnothing), & (\hat{u},\ \{(\texttt{h != o2})^0\})\big)\\
nodeValue(\texttt{h != o2}) &= \big((\hat{0}, \varnothing), & (\hat{u},\ \{h^1, h^0, o2^0\})\big)\\
weight\big((\texttt{h != o2})^0\big) &= \infty\\
nodeValue(\underbrace{o2 + 1}_{o3}) &= \big((\hat{u},\ \{o2^0\}), & (\hat{u},\ \{o2^0\})\big)
\end{aligned}$$

Third iteration of the loop, as the bits changed:

$$\begin{aligned}
nodeValue(\underbrace{\phi(o1, o3)}_{o2}) &= \big((\hat{u},\ \{(\texttt{h != o2})^0, o3^1\}), & (\hat{u},\ \{(\texttt{h != o2})^0, o3^0\})\big)\\
nodeValue(\texttt{h != o2}) &= \big((\hat{0}, \varnothing), & (\hat{u},\ \{h^0, h^1, o2^0, o2^1\})\big)\\
weight\big((\texttt{h != o2})^0\big) &= \infty\\
nodeValue(\underbrace{o2 + 1}_{o3}) &= \big((\hat{u},\ \{o2^0, o2^1\}), & (\hat{u},\ \{o2^0, o2^1\})\big)
\end{aligned}$$

Fourth iteration of the loop, as the bits changed:

$$\begin{aligned}
nodeValue(\underbrace{\phi(o1, o3)}_{o2}) &= \big((\hat{u},\ \{(\texttt{h != o2})^0, o3^1\}), & (\hat{u},\ \{(\texttt{h != o2})^0, o3^0\})\big)\\
nodeValue(\texttt{h != o2}) &= \big((\hat{0}, \varnothing), & (\hat{u},\ \{h^0, h^1, o2^0, o2^1\})\big)\\
weight\big((\texttt{h != o2})^0\big) &= \infty\\
nodeValue(\underbrace{o2 + 1}_{o3}) &= \big((\hat{u},\ \{o2^0, o2^1\}), & (\hat{u},\ \{o2^0, o2^1\})\big)
\end{aligned}$$

A fix-point is found, as no bits changed in the last iteration. The output o of the program equals $o2$. This results in the bit dependency graph presented in Figure 6.1. The marked bits h^1 and h^0 constitute a possible minimum-vertex-cut, as the conditional bit $(\texttt{h != o2})^0$ has an infinite weight. The approximated leakage for the program is 2 bits, which is correct, as the program leaks the whole secret. The analysis would have under-approximated the leakage as 1 bit if the setting of the weight for the conditional bit had been omitted.

7. Extension: Interprocedural Analysis

This chapter introduces an interprocedural version of the analysis that supports functions with multiple parameters and at most one return value. The main extension of the previously described analysis is that function calls are handled by a specific *function handler*.

Definition 7.1 (Function handler)**.** A function handler is a function that returns a function for every call-site that evaluates a specific call and returns the result, given the arguments of the call. This can be formalised as:

$$handler = CallSite \rightarrow (\underbrace{Seq[Value]}_{\text{arguments}} \rightarrow Value)$$

There are three main types of handlers that are described in the following: the *all* handler, the *inlining* handler and the *summary* handler.

7.1. *All* Function Handler

This is the most basic version of the interprocedural analysis. It yields a return value in which each bit has an unknown value and depends on all argument bits.

The handler can be formalised as:

$$\begin{aligned} all : CallSite &\rightarrow (Seq[Value] \rightarrow Value) \\ site &\mapsto \big(args \mapsto (\underbrace{(\hat{u}, \underbrace{\{b|b \in v, v \in args\}}_{=:\gamma}), \ldots, (\hat{u}, \gamma)}_{n \text{ times}})\big) \end{aligned}$$

The advantage of this handler is that it is trivially terminating for every function call. Its disadvantage is that it ignores the body of the called function and is therefore over-conservative compared to other handlers.

7.2. *Inlining* Function Handler

This handler is inspired by the call-string analysis technique described by Muchnik and Jones [35]. It is a basic version that inlines each function upon call. To avoid problems with recursion, each function is inlined only a fixed number of times per call-path. If this limit is reached, then another function handler, referred to as *bot-handler*, is used to obtain a result for the remaining call-sites.

7.3. *Summary* Function Handler

This handler pre-computes a summary-graph for every function that contains the dependencies between argument bits and return bits. This has the advantage of supporting unbounded finite recursion more precisely than the other handlers. The idea behind the calculation of the summary graphs is to evaluate each function with newly created, statically unknown argument bits and to then examine the dependencies between these arguments and the return value. This is repeated in a fix-point iteration on the call-graph. The *inlining* handler is used in each reevaluation, using the result of previous reevaluations as a bot-handler.

The resulting summary-graphs can be used when handling a call as follows: The summary graph for the specific function is cloned. The argument and return bits of the graph are then replaced by their actual correspondents at the call site.

7.3.1. Reduction

The inlining handler produces complex graphs. The graphs have to be reduced to get usable summary-graphs, as all graphs have to be cloned on call.

There are two different reductions: *basic* and *min-cut*.

***basic* reduction** The *basic* reduction removes all but the argument and the return bits, setting the dependencies of the return-bits to the argument bits that they have transitive dependencies to. This results in a reduction of precision, as the inner structure of the graph is omitted. An example for a result of this reduction is presented in Figure 7.3.

***min-cut* reduction** This reduction extends the *basic* reduction by including bits that constitute the minimum-vertex-cut, using the argument bits as a source and the return bits as a sink. The source and the sink bits have an infinite weight in this calculation. As a result, the distinct paths of all source and sink bits are included. This improves the precision, as the number of distinct paths between the argument bits and the return bits is not inflated compared to the *basic* reduction. An example for a result of this reduction is presented in Figure 7.5.

7.3.2. Fix-point iteration

The fix-point iteration iterates over the call graph. Each function node is mapped to the current summary graph for its function. The starting graph for each node has no connections between the argument and the return bits. The possible summary graphs for each function node form a bounded lattice, using the result of the *all* handler as a $\top$ element, the bit graph where the return bits have no dependencies as a $\bot$ element and the less relation on the number of connections between the argument and return bits.

```
int func(int a, int b){
  int r1 = 0
  if (a == 0) {
    r2 = b | 0b101
  }
  return ϕ(r1, r2) | 0
}
o = func(h | 1, l | 0)
```

Listing 7.1: Example of a program with a non-recursive function and three bit values

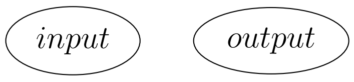


Figure 7.1.: Dependency graph for the example program given in Listing 7.1 using the inlining handler

7.4. Example

This illustrative example shows the differences of the function handlers for a non-recursive program for simplicity. The example program assumes three bit values and its code is presented in Listing 7.1. The following paragraphs give the analysis results, like the bit dependency graph, for an instantiation of the analysis using the inlining handler and the summary handler with the basic and the min-cut reduction. Bits that constitute a minimal-cut are marked.

7.4.1. Inlining handler

The inlining handler directly inlines the function call in the last line of the program and approximates the leakage as 0 bits, as the condition `a == 0` with $a \equiv$ `h|1` is never true. The resulting bit dependency graph for the program is given in Figure 7.1.

7.4.2. Summary handler

For the summary handler, the function `func` of the program is analysed before the actual analysis. The method graph produced by the internal inlining handler is given in figure Figure 7.2. This graph is then post-processed using the basic and the min-cut reduction, see Figure 7.3 and Figure 7.5. The resulting bit dependency graphs for the program are given in Figure 7.4 and Figure 7.6. The basic reduction reduces the method dependency graph to a smaller graph, but results in an approximated leakage of 2 bits. On the other hand the min-cut reduction results in a larger graph, but also in an approximated leakage of only 1 bit.

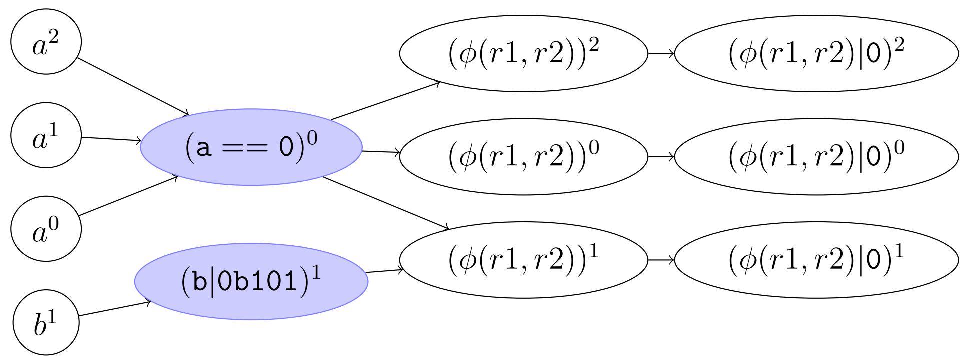

a^2
a^1
a^0
b^1
$(\texttt{a} == \texttt{0})^0$
$(\texttt{b|0b101})^1$
$(\phi(r1,r2))^2$
$(\phi(r1,r2))^0$
$(\phi(r1,r2))^1$
$(\phi(r1,r2)|\texttt{0})^2$
$(\phi(r1,r2)|\texttt{0})^0$
$(\phi(r1,r2)|\texttt{0})^1$

Figure 7.2.: Unreduced method dependency graph for the `func` function. The marked bits constitute the minimum-vertex-cut used by the min-cut reduction.

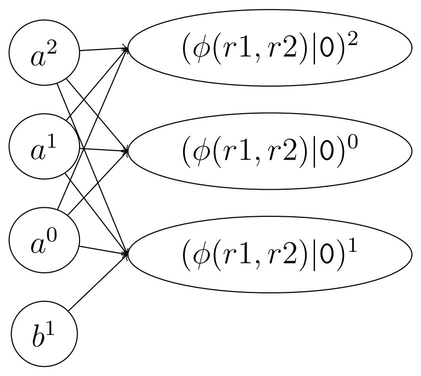

a^2
a^1
a^0
b^1
$(\phi(r1,r2)|\texttt{0})^2$
$(\phi(r1,r2)|\texttt{0})^0$
$(\phi(r1,r2)|\texttt{0})^1$

Figure 7.3.: Reduced version of the method dependency graph given in Figure 7.2 after applying the basic reduction

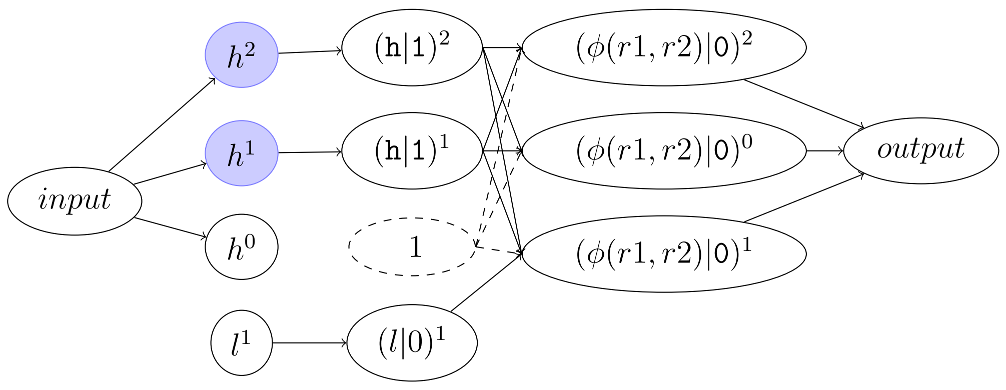

$input$
h^2
h^1
h^0
l^1
$(\texttt{h|1})^2$
$(\texttt{h|1})^1$
1
$(l|0)^1$
$(\phi(r1,r2)|\texttt{0})^2$
$(\phi(r1,r2)|\texttt{0})^0$
$(\phi(r1,r2)|\texttt{0})^1$
$output$

Figure 7.4.: Dependency graph of the program, analysed with the basic reduction summary handler. The dashed node is only included for readability.

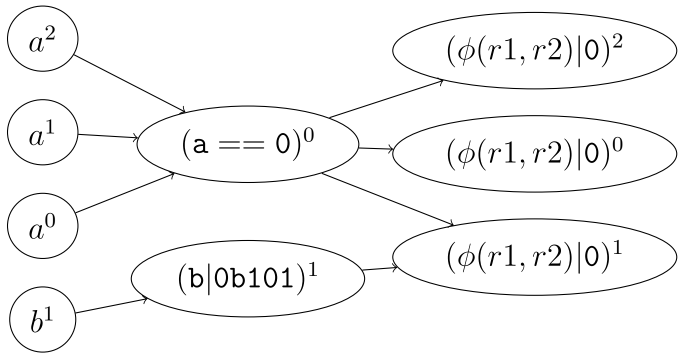

a^2
a^1
a^0
b^1
$(\texttt{a} == \texttt{0})^0$
$(\texttt{b}|\texttt{0b101})^1$
$(\phi(r1, r2)|\texttt{0})^2$
$(\phi(r1, r2)|\texttt{0})^0$
$(\phi(r1, r2)|\texttt{0})^1$

Figure 7.5.: Reduced version of the method dependency graph given in Figure 7.2 after applying the min-cut reduction

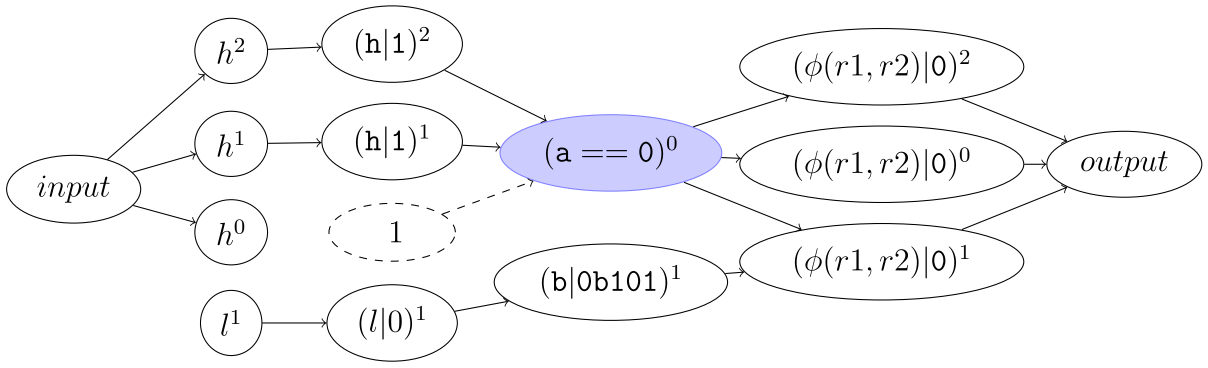

$input$
h^2
h^1
h^0
l^1
$(\texttt{h}|\texttt{1})^2$
$(\texttt{h}|\texttt{1})^1$
1
$(l|\texttt{0})^1$
$(\texttt{a} == \texttt{0})^0$
$(\texttt{b}|\texttt{0b101})^1$
$(\phi(r1, r2)|\texttt{0})^2$
$(\phi(r1, r2)|\texttt{0})^0$
$(\phi(r1, r2)|\texttt{0})^1$
$output$

Figure 7.6.: Dependency graph of the program, analysed with the min-cut reduction summary handler. The dashed node is only included for readability.

8. Implementation

The analysis has been implemented in two tools: as a stand-alone demo implementation with a graphical user interface[1] and on top of JOANA [2]. The stand-alone demo implementation is the first experimental version that was used for debugging and preliminary testing the analysis ideas and the generation of evaluation programs, see section 9.3. We then implemented the final version of the analysis on top of JOANA, learning from the mistakes in the first implementation. The JOANA-based implementation is therefore technically much more mature and stable than the stand-alone demo implementation.

Both implementations are developed in a project called *Nildumu*[2] project. These implementations are prototypes written in Java 8 and limited to the constructs of a while-language with functions.

8.1. Stand-Alone Demo Implementation

The stand-alone demo implementation is based on an LR-parser and lexer generator developed before this work. This implementation of the analysis has its own programming language that is restricted to language features supported by the analysis. Programs written in this language can be analysed using a graphical user interface which plots the bit dependency graphs.

The programming language is C-like and supports only `int` as a data type and functions with one return value and multiple parameters. The `return` statement of a function has to be the last statement in the function. The code inside a function cannot access any global variables. In the global scope, variables can be declared as `output` and `input` on varying security levels. The only supported control structures are `while` and `if` (with an optional `else`). Integers are implicitly converted into boolean by only using their lowest significant bit. The assumed width of an `int` can be configured.

Furthermore, the analysis works on an extended abstract syntax tree in SSA-form. Although the used data structure is not a proper PDG, the analysis works, with a few exceptions, the same as for a PDG. The advantage of this data structure is that it results in a more direct mapping between nodes and source code, making it easier to understand.

The major disadvantage is that this pseudo-PDG implementation is hard to maintain. It requires a custom SSA transformation and other basic analysis steps

[1] https://github.com/parttimenerd/nildumu

[2] *Nildumu* is lojban for "something is a quantity".

that are already included in JOANA.

8.2. JOANA-Based Implementation

The implementation[3] based on JOANA uses the underlying WALA framework extensively and works on a proper PDG implementation, provided by JOANA. This is a major advantage, as the creation of PDGs is error-prone. Another advantage is that the underlying infrastructure includes support for an object-sensitive analysis and more Java features which could be useful in the future, see Section 10.2.

The major disadvantage is that the PDG does not contain information on the actual operators as JOANA is focussed on Qualitative Information Flow. We circumvent this lack of information by using the underlying bytecode instructions and basic block structures. Especially the correct implementation of the fix-point iteration and the gathering of parameter conditions for ϕ-nodes was difficult and error-prone.

[3]https://github.com/parttimenerd/nildumu-joana

9. Evaluation

In this chapter, three different Quantitative Information Flow analysis tools are compared with Nildumu. The JOANA-based implementation of the Nildumu project is used in the following, as it is technically more mature. The analysis tools are compared using standard benchmarks from literature. The analysis tools are described in Section 9.1, followed by the benchmark programs in Section 9.2, the evaluation process in Section 9.3 and the evaluation results Section 9.4.

9.1. Compared Analyses

The three analysis tools that are compared with Nildumu[1] are examples of the different categories of analyses introduced in the introduction chapter: ApproxFlow is based on a static approximation, Flowcheck and LeakWatch on dynamic observation. More tools were considered, but we were unable to access or successfully run them.

ApproxFlow [9] This tool creates a SAT formula that represents the program using the bounded model generator CBMC [36]. The tool then post-processes the SAT formula and uses the approximate model counter ApproxMC2 [37] to approximate the number of different possible outputs of the program. ApproxFlow implements a theoretically sound static analysis that produces an approximation of the leakage. The arguments for CBMC cannot be altered in the original command-line interface of the tool, the evaluation therefore uses a modified version[2].

Flowcheck [14] This tool is based on the `memcheck` tool of valgrind[3] and uses dynamic tainting to dynamically approximate the leakage of certain program evaluations. The basic concept behind this tool is the same as the concept behind Nildumu: The tool tracks the dependencies between the bits and uses a minimum–cut algorithm to approximate the leakage. The main difference is that the tool only approximates the leakage for a given input and does not return a conservative approximation for all inputs. Flowcheck fully supports any binary and works with programs with several hundred thousand lines of code.

We developed a tiny wrapper that calls the tool multiple times with different random inputs and takes the maximum of the returned leakages to be able to compare

[1]The evaluated version of the stand-alone demo implementation of Nildumu is `49d3eef`, the evaluated version of the JOANA-based implementation is `313e8d2`.
[2]The modified version of ApproxFlow can be found at https://github.com/parttimenerd/approxflow
[3]http://valgrind.org/

this tool with the other static analysis tools. This wrapper calls the tool three times in a row and repeats this as long as the maximum leakage does change. It also ends the analysis if the tool returns the same leakage three times in a row. This reduces the variance of the leakage approximation. As a result of using this wrapper, the results for Flowcheck in this work might vary from other evaluations in literature.

LeakWatch [12] This tool, version 0.5 is used here, supports analysing arbitrary Java code in a dynamic manner. The program that is to be analysed is run several times, with its secret inputs and public output observed. The resulting input-output-pairs are then statistically analysed to get an approximation of the leakage. The inputs have to be generated at runtime. The evaluation in this chapter uses the `SecureRandom` class of Java to generate random secrets.

It differs from the Flowcheck tool by automatically reexecuting the program until the computed leakage is stable. The other difference is that it does not use dynamic tainting but works solely by observing the inputs and outputs.

9.2. Benchmark Programs

This section describes the benchmark programs used in the evaluation. The benchmarks from literature are obtained from [38], [39] and [7]. These benchmarks are used in many other papers to compare different analyses with each other. The following describes the benchmark programs including their leakage.

The programs are given in the format of the previous examples and are then transformed into the input formats of each analysis as the input format for the compared analyses varies. The benchmarks only use a common subset of the features that all tools support, except for low inputs that are not supported by ApproxFlow. Nildumu only supports a while-language with functions, limiting the complexity of the benchmark programs.

All listed programs assume that the variable `h` is a high input, `l` a low input, `o` a low output and that values are 32 bits wide.

9.2.1. Benchmarks from literature

Binary search This program splits its input into a sum of powers of two, dropping all powers of two lower than 2^{16} and outputting the sum. It should leak the first 16 bits of the input. The program was first described by Meng et al. [38], its code is given in Listing 9.1.

Electronic purse This program was originally presented by Backes et al. [39]. In this program `h` represents the balance of a bank account and the output the number of times that 5 units of money can be withdrawn. The balance is limited to 20 units. The program should leak 2 bits of information and its code is given in Listing 9.2.

```
int BITS = 16
int z = 0
for (int i = 0; i < BITS; i++) {
  int m = 1 << (31 - i)
  if (z + m <= h) {
    z = z + m
  }
}
o = z
```

Listing 9.1: Binary search

```
int z = 0
while (h >= 5 && h < 20) {
  h = h - 5
  z = z + 1
}
o = z
```

Listing 9.2: Electronic purse

```
int z = h & 0x77777777
int x
if (z <= 64) {
  x = z
} else {
  x = 0
}
if (x % 2 == 0) {
  x = x + 1
}
o = x
```

Listing 9.3: Illustrative example

Illustrative example This program is used for illustration purposes by Meng et al. [38]. It leaks $\log_2 17 \approx 4.1$ bits and is presented in Listing 9.3.

Implicit flow This program, as described by Meng et al. [38], is a cascade of if-statements that produces 7 different outputs and therefore leaks $\log_2 7 \approx 2.8$ bits. The code of this program is given in Listing 9.4.

Masked copy This program was first presented by Meng et al. [38] and leaks the upper 16 bits of the secret. The code of this program is given in Listing 9.5.

Mix and duplicate This program splits the secret into two halves. Both halves are then XORed. The result of this operation is then split into two halves. The upper half is ORed with the lower half. The result of this operation is appended to the lower half and returned. This should, according to Newsome et al. [7], present problems for tools that examine the whole input space. The program leaks 16 bits and its code is presented in Listing 9.6.

Population count This program, first described by Newsome et al. [7], counts the number of set bits in the input. It leaks $\log_2 33 \approx 5.0$ bits and its code is given in Listing 9.7.

Password checker This is the typical introductory program for information flow that checks whether the input `l` is the correct password `h`. It should leak 1 bit. The code of this program is given in Listing 9.8.

Sanity check This program limits the high input and the output to a certain range, possibly posing problems to dynamic analyses [7] and leaking 4 bits. Typical values

```
int z = 0
if (h == 0) {
  z = 0
} else if (h == 1) {
  z = 1
} else if (h == 2) {
  z = 2
} else if (h == 3) {
  z = 3
} else if (h == 4) {
  z = 4
} else if (h == 5) {
  z = 5
} else if (h == 6) {
  z = 6
} else {
  z = 0
}
o = z
```

Listing 9.4: Implicit flow

```
o = h & 0xffff0000
```

Listing 9.5: Masked copy

```
int z = ( ( h >> 16 ) ^ h ) & 0x0000ffff
o = z | ( z << 16 )
```

Listing 9.6: Mix and duplicate

```
int z = ( h & 0x55555555 ) + ( ( h >> 1) & 0x55555555 )
z = ( z & 0x33333333 ) + ( ( z >> 2) & 0x33333333 )
z = ( z & 0x0f0f0f0f ) + ( ( z >> 4) & 0x0f0f0f0f )
z = ( z & 0x00ff00ff ) + ( ( z >> 8) & 0x00ff00ff )
o = ( z + (z >> 16 ) ) & 0x0000ffff
```

Listing 9.7: Population count

```
int z
if (h == 1) {
  z = 1
} else {
  z = 0
}
o = z
```

Listing 9.8: Password checker

```
int z
if (h >= 0 && h < 16) {
  z = base + h
} else {
  z = base
}
o = z
```

Listing 9.9: Sanity check

for `base` are `0x00001000` and `0x7ffffffa`. The code of this program is presented in Listing 9.9.

Sum This program sums up three different high inputs and should leak 32 bits, it was first presented by Backes et al. [39]. The code of this program is given in Listing 9.10.

```
o = h + i + j
```

Listing 9.10: Sum

9.2.2. Recursion related benchmarks

The following benchmark programs test the ability of the analyses to work with recursion properly.

Fibonacci This program is a basic implementation of the Fibonacci function. This program leaks $\log_2 31 \approx 5.0$ bits for version 1 and $\log_2 1023 \approx 10.0$ bits for version 2.

```
int fib(int num) {
  int r = 1
  if (num > 2) {
    r = fib(num - 1) + fib(num - 2)
  }
  return r
}
// version 1:
o = fib(h & 31)
// version 2:
o = fib(h & 1023)
```

Listing 9.11: Fibonnaci

```
int fib(int num) {
  int r = 1
  if (r > 2) {
    r = fib(num - 1) + fib(num - 2)
  }
  return r
}
o = fib(h & 31)
```

Listing 9.12: Dead Fibonacci

It poses problems to analyses, as the number of function calls increases non-linearly with the input parameter. The code of this program is presented in Listing 9.11.

Dead Fibonacci This program is a variant of the *Fibonacci* program that returns 1 for all inputs and thereby leaks no information on the secret input. It might cause problems for static analyses, as the function `fib` recursively calls itself in unreachable code. The code of this program is given in Listing 9.12.

Recursive id This is a simple program with linear recursion which contains an *id* function that returns the value of its parameter for positive parameters and zero otherwise. The program results in a leakage of 31 bits. Its code is presented in Listing 9.13.

Dead recursive id This program is an adaption of the *Recursive id* program that returns 0 for every input, resulting in a leakage of 0 bit. The code of this program is

```
int id(int num) {
  int r = 0
  if (num > 0) {
    r = id(num - 1) + 1
  }
  return r
}
o = id(h)
```

Listing 9.13: Recursive id

```
int id(int num) {
  int r = 0
  if (num > 0) {
    r = id(num - 1) + 1
  }
  return 0
}
o = id(h)
```

Listing 9.14: Dead recursive id

presented in Listing 9.14.

9.2.3. If-statement scalability benchmark

This benchmarking program is used to evaluate the scalability of the analyses for programs of increasing size. The size of the program given in Listing 9.15 can be varied by altering α. The program has α different outputs and therefore leaks $\log_2 \alpha$ bits.

9.3. Evaluation Process

The following describes the evaluation process and environment. This is based on the guidelines from our previous work [40] and uses the therein developed tool *temci*[4] to actually benchmark. The evaluation framework itself is implemented in the stand-alone demo Nildumu tool.

[4]https://temci.readthedocs.io/en/latest/

```
int z = 0
if (h == 1) {
  z = 1;
}
// ...
if (h == α) {
  z = α;
}
o = z
```

Listing 9.15: If-statement scalability benchmark

Analysis tools The analysis tools are benchmarked partly in different configurations, to gather results even when the original configuration resulted in a non-termination of the analysis. ApproxFlow is evaluated in three configurations, depending on the amount of loop unrolling and function inlining. The original configuration unrolls and inlines 32 times. The other configurations unroll and inline 2 and 5 times.

Both implementations of Nildumu are evaluated: The JOANA-based implementation is evaluated in two configurations, depending on the level of inlining for the used inlining handler and the level of inlining in the summary handler, called by the inlining handler: the first configuration inlines 2 times, the second configuration 5 times in both cases. The stand-alone demo implementation of Nildumu is only evaluated in the configuration that inlines 2 times. In the following the term *Nildumu* refers to the much more mature JOANA-based implementation.

Flowcheck is evaluated in two configurations: with and without the calculation of the minimum-cut. Enabling the calculation of the minimum-cut increases the precision of the analysis, but we were unable to use it in the main benchmarking environment, as it resulted in the analysis aborting with the following error message: “t5 = CASle(t10::t3->t4) Flowcheck: the ’impossible’ happened: flowcheck trace: unhandled IRStmt”.

Environment The main evaluation is done on a desktop computer with 64 GiB of RAM and a hyper-threaded Intel Core i7-6700 CPU with 4 physical cores. This computer has a SSD with 240 GiB as its hard-drive and runs an Ubuntu 16.04.5 LTS with kernel version 4.4.0-137-generic without a graphical user interface. The CBMC version on this computer is 5.3.

Flowcheck with enabled minimum-cut calculation was evaluated on a laptop with 8 GiB of RAM, a hyper-threaded Intel Core i5-4300U CPU with 2 physical cores and a 500 GiB SSD as its hard-drive. This computation runs a Debian Buster with GNOME and kernel version 4.18.0-2-amd64 and is in the following referred to as the second benchmarking environment.

Benchmarking The benchmarks were gathered by executing each analysis 10 times, using the GNU time utility[5] for the actual measurements and taking their mean. The recorded time includes the runtime of all pre-processing tools that an analysis needs. For example, the Flowcheck tool works on binaries, therefore the runtime includes the compilation of the analysed code with the GNU C compiler[6], version 5.4.0 in the main and 8.2.0 for the second benchmarking environment.

9.4. Results

This section presents the individual results of all analyses and their discussion. The timeout for the benchmarks was 30 minutes, timeouts are marked as -. The standard deviations were below 5% unless marked otherwise. If the runtimes or approximated leakages for different configurations of the same analysis differ less than one standard deviation, then the mean is presented. Under-approximations of programs are marked as bold and underlined.

9.4.1. Benchmarks from literature

The runtimes are presented in Table 9.3, the leakages in Table 9.2.

ApproxFlow The benchmarking results for ApproxFlow seem to be unpredictable, especially for programs like *Binary Search*: The amount of loop unrolling is not related to the performance. The reason for this might be that modern SAT solvers, on which ApproxFlow is based, use complex heuristics resulting in widely varying runtimes for similar formulas [41]. This is the reason why ApproxFlow runs faster for an unrolling level of 32 compared to an unrolling level of 5 although the latter results in a smaller SAT formula.

Another big problem of ApproxFlow is that it does support functions only by inlining them multiple times, resulting in long runtimes for small programs even if the functions are actually never called during a program's execution, see *Dead Fibonacci*. Besides the long runtimes, it can also cause under-approximations if functions are called more often than they are inlined. Other approaches for supporting functions in bounded model checking based tools are presented in Section 10.2.2.

Loop unrolling is also a cause of under-approximations, as can be seen for the *Electronic purse* program. The loop body in this program is executed at most 4 times. If the loop is unrolled only 2 times, then the effects of the two other possible iterations are ignored, leading to the observed under-approximation.

[5] https://www.gnu.org/software/time/
[6] https://www.gnu.org/software/gcc/

Program	ApproxFlow			Flowcheck	LeakWatch	Nildumu		
	2	5	32			2	5	demo[3]
Binary Search	0.788	0.812	0.574	1.153	-	2.758		**error**[3]
Electronic Purse	0.129	0.140	0.241	1.114	-	1.735		0.427
Illustrative ex.		0.882		1.141	-	1.637		0.344
Implicit Flow		0.129		1.153	-	1.573		0.357
Masked Copy		0.270		2.272***	-	1.554		0.310
Mix and dup.		0.280		1.731***	-	1.566		0.328
Pass. checker		**unsupported**[1]		1.200	-	1.587		0.321
Pop. count		0.200		1.148	-	1.635		0.332
Sanity check (1)		0.811		1.151	-	1.615		0.344
Sanity check (2)		1.093		1.152	-	1.603		0.349
Sum		0.801		1.945***	-	1.581		0.352
Dead Fibonacci		0.774		1.150	-	1.611		0.344
Fibonacci (1)	0.947	4.710	-[2]	54.433**	-	3.480	67.332	0.799
Fibonacci (2)	0.940	5.789	-	-	-	3.475	62.693*	0.824
Dead rec. id	0.828	0.933	2.637	3.679***	-	1.625		0.378
Recursive id	0.829	0.935	2.624	2.346***	-	2.013	2.487*	0.526

Table 9.1.: The runtimes of the analyses in seconds.

[1] ApproxFlow does not support low inputs.

[2] ApproxFlow did not terminate after 16 hours, as the model generator CBMC was still generating the formula.

[3] The stand-alone demo version of Nildumu does not support this test case correctly

* The standard deviation is between 5% and 10%.

** The standard deviation is between 10% and 30%.

*** The standard deviation is between 30% and 90%.

Program	C	ApproxFlow			Flowcheck[1]	LeakWatch	Nildumu[2]
		2	5	32			
Binary Search	16.0	32.0	32.0	15.0	16.0	-	32.0[3]
Electronic Purse	2.0	**1.0**	2.0	2.0	2.0	-	32.0
Illustrative ex.	4.1		30.0		**1.0**	-	32.0
Implicit Flow	2.8		2.8		7.0	-	3.0
Masked Copy	16.0		16.0		334.0	-	16.0
Mix and duplicate	16.0		16.0		331.0	-	16.0
Password checker	1.0		**unsupported**		1.0	-	1.0
Population count	5.0		8.2		200.0	-	10.0
Sanity check (1/2)	4.0		31.0		**1.0**	-	32.0
Sum	32.0		32.0		334.0	-	32.0
Dead Fibonacci	0.0		32.0		0.0	-	0.0
Fibonacci (1)	5.0	32.0	32.0	-	2692537.0	-	5.0
Fibonacci (2)	10.0	32.0	32.0	-	-	-	10.0
Dead recursive id	0.0		32.0		32.0	-	0.0
Recursive id	31.0		32.0		32.0	-	32.0

Table 9.2.: The approximated leakage for the basic benchmark program calculated by the different analyses in bits rounded to the first decimal digit. The calculated leakages did not vary between the different runs of the analyses. The second column C gives the actual leakages of the programs.

[1] Flowcheck is called with disabled minimum-cut calculation, this reduces the precision, see Table 9.4.

[2] All evaluated implementations and configurations of Nildumu resulted in the same computed leakage.

[3] The stand-alone demo version of Nildumu does not support this program correctly.

Program	Flowcheck		Nildumu
	with mininum-cut	without minimum-cut	2
Binary Search	2.360	0.931	5.085
Electronic Purse	2.348	0.929	3.659
Illustrative ex.	2.355	0.969	2.952
Implicit Flow	2.360	0.946	2.880
Masked Copy	2.359	1.376***	2.717
Mix and duplicate	2.358	1.521***	2.804
Password checker	2.356	0.929	2.860
Population count	2.363	0.944	3.061
Sanity check (1/2)	2.356	0.942	2.927
Sum	2.372	1.936	2.880
Dead Fibonacci	2.357	0.928	2.847
Fibonacci (1)	-	71.194***	6.004
Fibonacci (2)	-	-	6.000
Dead recursive id	3.180***	2.429***	2.933
Recursive id	2.754**	2.931***	3.721

Table 9.3.: The runtimes of the two different configurations of Flowcheck and the JOANA-based implementation Nildumu in seconds. These measurements were obtained in the second benchmarking environment.
** The standard deviation is between 10% and 30%.
*** The standard deviation is between 30% and 90%.

Program	C	Flowcheck with minimum-cut	Flowcheck without minimum-cut	Nildumu
Binary Search	16.0	16.0	16.0	32.0
Electronic Purse	2.0	2.0	2.0	32.0
Illustrative ex.	4.1	**1.0**	**1.0**	32.0
Implicit Flow	2.8	7.0	7.0	3.0
Masked Copy	16.0	16.0	334.0	16.0
Mix and duplicate	16.0	16.0	331.0	16.0
Password checker	1.0	1.0	1.0	1.0
Population count	5.0	10.0	200.0	10.0
Sanity check (1/2)	4.0	**1.0**	**1.0**	32.0
Sum	32.0	32.0	334.0	32.0
Dead Fibonacci	0.0	0.0	0.0	0.0
Fibonacci (1)	5.0	-	2692537.0	5.0
Fibonacci (2)	10.0	-	-	10.0
Dead recursive id	0.0	5.0	32.0	0.0
Recursive id	31.0	5.0	32.0	32.0

Table 9.4.: The approximated leakage for the basic benchmark program by the two different configurations of Flowcheck and the JOANA-based implementation of Nildumu. The calculated leakages did not vary between the different runs of the analyses. The second column C gives the actual leakages of the programs.

LeakWatch This tool did not terminate for any program after 30 minutes: The problem is that the input space consists of 2^{32} elements.

LeakWatch also has problems with programs that leak a high amount of information, as the tool is then unable to decide whether the obtained result is stable and observing a high number of outputs needs many reexecutions. The tool therefore is only usable for smaller input spaces, terminating for all programs for a bit width of for example 5 bits.

Flowcheck The dynamic nature of Flowcheck results in under-approximations for programs, like *Illustrative example*, as only a few inputs are evaluated. The under-approximations are infrequent, as the analysis treats control-flow-dependencies properly. The proper treatment of control-dependencies resulted in an over-approximation for the *Implicit flow* program which only leaks due to control dependencies. Enabling the minimum-cut calculation significantly improves the precision of the analysis, see for example the *Masked copy* program for which the approximated leakage decreased from 334 to 16 bits in Table 9.4. But the minimum-cut version has also double the runtime for most examples.

Nildumu The JOANA-based version of Nildumu is needs to load the complex JOANA framework consisting of over 400 MiB of Java JARs before a program can be analysed. This is the cause for the high runtimes even for small programs like *Password checker* and why the runtime is roughly the same for most programs.

The stand-alone demo version of Nildumu has far less dependencies and processes the program code directly without compiling it to bytecode and building up a proper PDG. This version is therefore much faster but also less mature, e.g. it cannot analyse the *Binary Search* program correctly.

A major problem of the analysis as whole is that loops and conditions are treated conservatively. Nildumu can handle arbitrary recursion by using summary-graphs with good precision and performance, compared to the other tools.

9.4.2. If-statements scalability test

Nildumu and LeakWatch failed for programs with more than 2^{12} if-statements, due to constraints of the JVM that limits the size of methods and number of constants in class files [42, §4.11]. ApproxFlow ran into a timeout for programs larger than 2^{14} if-statements. This is the reason why the results for programs with more than 2^{14} if-statements are not presented in the following. The runtimes are given in Table 9.5 and plotted in figure 9.1, the calculated leakages in Table 9.6. The two configurations of Flowcheck are evaluated in the second benchmarking environment, the results are given in Table 9.7 and Table 9.8 and compared with the JOANA-based implementation of Nildumu.

$\log_2(\lvert\text{if-stmts}\rvert)$	ApproxFlow	Flowcheck	LeakWatch	Nildumu	N. Demo[1]
0	**error**	1.139	-	1.523	0.317
1	**error**	1.143	-	1.537	0.331
2	0.120	1.141	-	1.550	0.332
3	0.129	1.140	-	1.605	0.354
4	0.131	1.140	-	1.651	0.370
5	0.150	1.143	-	1.771	0.409
6	0.519	1.151	-	1.943	0.456
7	1.372	1.160	-	2.311	0.536
8	4.843	1.178	-	2.944	0.650
9	18.916	1.213	-	4.340	0.851
10	49.369	1.281	-	7.648	1.161
11	110.119	1.423	-	18.639	1.672
12	125.488	1.708	-	59.965	2.560
13	174.241	2.299	**error**	**error**	4.444
14	583.775	3.498	**error**	**error**	7.791

Table 9.5.: Runtime of the scalability test in seconds. The standard deviations for all measurements are below 6%. Most of the runtime for the smaller programs is related to the starting of the analysis tools.
[1] Stand-alone demo version of Nildumu

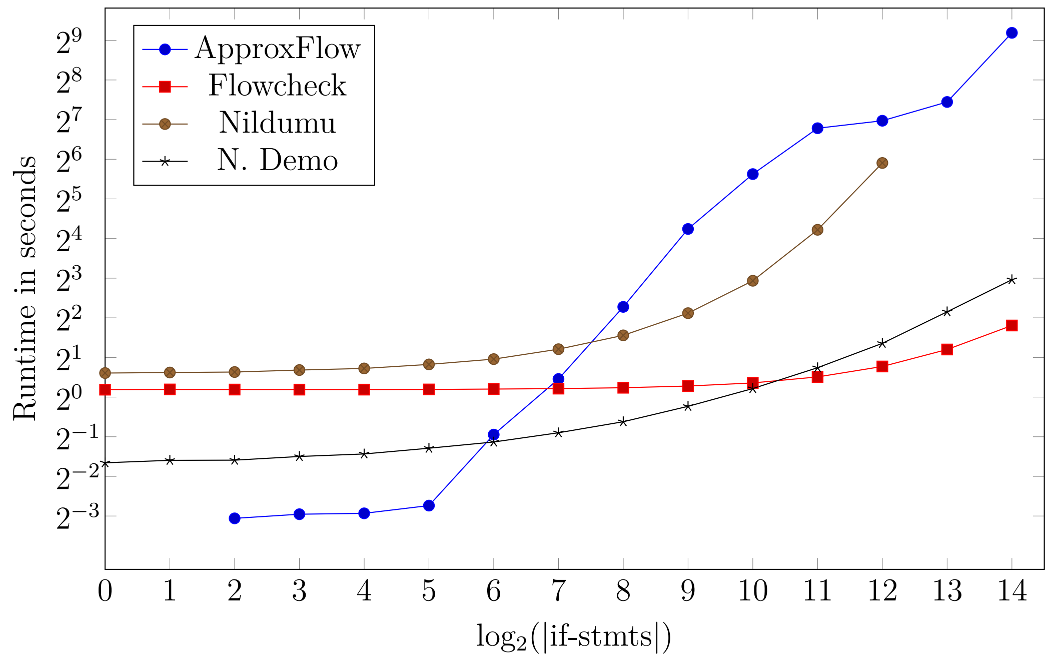


Figure 9.1.: Runtime of the scalability test in seconds.

$\log_2(\lvert\text{if-stmts}\rvert)$	C	ApproxFlow	Flowcheck	LeakWatch	Nildumu	N. Demo
0	0	**error**[1]	1	-	0	0
1	1	**error**[1]	2	-	1	1
2	2	2	4	-	3[2]	3[2]
3	3	3	8	-	4	4
4	4	4	16	-	5	5
5	5	5	32	-	6	6
6	6	6	64	-	7	7
7	7	7	128	-	8	8
8	8	8	256	-	9	9
9	9	9	512	-	10	10
10	10	10	1024	-	11	11
11	11	11	2048	-	12	12
12	12	12	4096	-	13	13
13	13	13	8192	**error**[3]	**error**[3]	14
14	14	14	16384	**error**[3]	**error**[3]	15

Table 9.6.: The approximated leakage for the scalability test in bits.

[2] Three bits of the output are statically unknown as the output ranges from 0b0001 to 0b0100, therefore the leakage is 3 for Nildumu.

[3] The size of the program exceeds the capabilities of the JVM as mentioned before.

$\log_2(\lvert\text{if-stmts}\rvert)$	Flowcheck		Nildumu
	with minimum-cut	without minimum-cut	
0	2.353	0.937	2.700
1	2.366	0.930	2.778
2	2.393	0.942	2.774
3	2.358	0.944	2.859
4	2.364	0.948	2.961*
5	2.381	0.950	3.070
6	2.412	0.953	3.465
7	2.464	0.976	3.874
8	2.577	1.001	4.674
9	2.793	1.047	6.321
10	3.239	1.153	10.755
11	4.151	1.373	23.836
12	6.004	1.847	154.992
13	9.731	2.769	**error**
14	17.275	4.793	**error**

Table 9.7.: The runtimes of the two different configurations of Flowcheck and the JOANA-based implementation of Nildumu in seconds. These measurements were obtained in the second benchmarking environment.
* The standard deviation is between 5% and 10%.

$\log_2(\lvert\text{if-stmts}\rvert)$	C	Flowcheck		Nildumu
		with minimum-cut	without minimum-cut	
0	0	1	1	0
1	1	2	2	1
2	2	4	4	3
3	3	8	8	4
4	4	8	16	5
5	5	8	32	6
6	6	8	64	7
7	7	8	128	8
8	8	8	256	9
9	9	$\underline{8}$	512	10
10	10	$\underline{8}$	1024	11
11	11	$\underline{8}$	2048	12
12	12	$\underline{8}$	4096	13
13	13	$\underline{8}$	8192	**error**
14	14	$\underline{8}$	16384	**error**

Table 9.8.: The approximated leakage for the scalability test in bits per analysis for the two configurations of Flowcheck and the JOANA-based implementation Nildumu, evaluated in the second benchmarking environment.

Precision ApproxFlow is the most precise tool, followed by Flowcheck and Nildumu. LeakWatch does not terminate, as before, even after 30 minutes. Both implementations of Nildumu slightly over-approximate the leakage because they count varying bits.

Runtimes A major part of the runtimes for smaller programs is the start-up time for the analysis tools and its pre-processor. The runtime of ApproxFlow increases approximately linear in the number of if-statements in the range of 2^5 to 2^{11} if-statements. The runtime of the stand-alone demo implementation of Nildumu increases sub-linearly and is the fastest analysis tool for programs with 2^6 to 2^{10} if-statements.

Flowcheck Flowcheck treats conditions conservatively and as a result the calculated leakages are overly conservative for programs with many conditions. The dynamic nature of Flowcheck leads to under-approximations due to the limited number of observed inputs. The tracking of the bit-dependencies and the minimum-cut calculation dominates the overall for runtime even for small programs but increases the precision and leads to under-approximations for larger programs.

9.4.3. Conclusion

All evaluated analysis tools besides Nildumu have problems with recursive functions and can produce under-approximations. Nildumu is less performant than the other tools, partly due to overhead of loading JOANA, but it scales well for larger programs and the performance is comparable to Flowcheck with enabled minimum-cut calculation. Furthermore, Nildumu analyses the presented benchmark programs with a precision that is comparable to other tools for some programs like *Implicit Flow* or *Population count* and better for programs with recursion. All tools have, overall, different strengths and weaknesses: ApproxFlow works well for programs without recursion. Nildumu works well for unbounded recursion, but has problems with loops. Flowcheck is by design prone to under-approximations. LeakWatch only works for small input spaces.

10. Conclusion und Future Work

10.1. Conclusion

The analysis presented in this thesis is a Quantitative Information Flow analysis that supports a while-language extended with functions. This is the first analysis, to our knowledge, that supports recursion properly. It adapts the well-known concept of summary edges. The analysis is, to our knowledge, the first data flow analysis in this field, see Appendix B.

The current version of the analysis is minimal. It does not support objects, arrays or other data types than integer and only works well for bit-operations. Being compact and based on a typical data flow analysis has the advantage that these features can later be added easily, as explained in the next section. The evaluation results presented in Section 9.4 show that the analysis produces comparably good results for typical examples with a comparable performance.

The analysis has in its current state several disadvantages compared to other analyses: The analysis is hard to implement efficiently, as many bit objects have to be created. Loops are approximated in an overly conservative manner as the used bit lattice is not well suited for comparison operators. The summary graphs are limited, as the summary function handler assumes that all parameter bits are statically unknown and not as placeholders for concrete bits. The thesis presented here is a condensed version of the algorithm, as many of the proposed improvements need more time to mature.

But nonetheless, the analysis presents, in our opinion, a good foundation for other analyses in the field of Quantitative Information Flow.

10.2. Future Work

This section gives an outlook on possible improvements and usages of the analysis presented in this thesis that could be explored in the future.

10.2.1. Improvements of the analysis

We expect that the analysis can be extended using the techniques already developed for data flow analysis. This includes techniques to gain object-sensitivity or support arrays. Other improvements could come from the integration of typical compiler optimisations like loop unrolling or common sub-expression elimination. It might furthermore be useful to improve the conditional knowledge propagation of Chapter 5 by adapting algorithms described in recent papers like [43], which use SMT and SAT solvers to increase precision.

Interval lattice Another possibility to improve the analysis, orthogonal to the others, is to choose a different lattice instead of the bit lattice. The problem of the bit lattice is that is best suited for analysing bitwise-operations, but not arithmetic operations [44]. An interesting lattice could be the interval-based lattice described in the following. The idea is to split each value into multiple intervals instead of n bits.

$$\mathit{Value} = \mathcal{P}(\mathit{BitInterval})$$

These intervals are comparable to the bits in the original analysis with the addition of constraints and the possibility of overlapping:

$$\begin{aligned} \mathit{BitInterval} &= \mathit{Identity} \times \underbrace{\mathcal{P}(\mathit{Identity})}_{\text{dependencies}} \times \mathcal{P}(\mathit{Constraint}) \\ \mathit{Identity} &= \mathit{Node} \times \underbrace{\mathbb{Z}}_{\text{lower bound}} \times \underbrace{\mathbb{Z}}_{\text{upper bound}} \end{aligned}$$

The problem with these lattices is that the number of intervals is $\frac{2^n(2^n-1)}{2}$ in the worst case. Therefore merging intervals of the same value is necessary during the analysis. A set of constraints like "i$^{\text{th}}$ bit is zero" could be used to partially counter the detrimental effect of merges to the precision of the analysis. The constraints could also be used to handle bit-operations properly. The difficulty is to design the constraints properly and to develop fast heuristics for selecting the intervals to be merged.

The leakage calculation might pose a problem as the new interval dependency graph is more complex than the original bit-dependency graph. Therefore, the transformation into an optimisation problem with poly-time solvers is much more difficult.

10.2.2. Usage of the analysis

This sections details two possible uses of the analysis and its possibly improved version.

Implementation in compilers As the developed analysis is essentially an extended constant-bit analysis, it would be interesting to implement it in a real compiler to improve the constant propagation as compilers like GCC only implement sparse conditional bit constant propagation [45].

SMT-based model counting A typical approach in Quantitative Information Flow Control is to approximately count the models of a SAT formula that represents the program that should be analysed. An analysis of this kind is described in Section 9.1.

Using SAT formulas has the disadvantage that dealing with loops and recursion is difficult and often realised with unrolling and inlining. Recent advances in the field of approximate SMT model counting [46] could make it possible to use SMT formulas instead.

The advantage of using SMT over SAT is that there are algorithms to generate SMT formulas that support loops and recursion properly. Loops can be analysed using a technique called *k-induction* [47] and recursion can be modelled using model based projection and interpolation techniques [48]. This is implemented in tools like SeaHorn [49].

The idea is to use a fast but imprecise analysis, like the analysis presented in this thesis, to get a rough estimate of the leakage of a program and use this number to improve the speed of the approximate SMT-model counting [46]. Another approach is to use the information from the faster analysis to pre-process the SMT formula. Such information could be that some bits are constant or that a value lies in a given range.

Bibliography

[1] H. Mantel, “Information Flow Control and Applications — Bridging a Gap —,” in *FME 2001: Formal Methods for Increasing Software Productivity* (J. N. Oliveira and P. Zave, eds.), (Berlin, Heidelberg), pp. 153–172, Springer Berlin Heidelberg, 2001.

[2] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, and D. Wasserrab, “Checking Probabilistic Noninterference Using JOANA,” *it - Information Technology*, vol. 56, pp. 280–287, nov 2014.

[3] M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein, “BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations,” in *Proceedings of the 2000 Europar Conference*, vol. 1900 of *Lecture Notes in Computer Science*, (Munich, Germany), pp. 969–979, Springer Verlag, aug 2000.

[4] G. Lowe, “Quantifying Information Flow,” in *15th IEEE Computer Security Foundations Workshop*, pp. 18–31, 2002.

[5] G. Smith, “On the Foundations of Quantitative Information Flow,” in *12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS’09)* (L. de Alfaro, ed.), vol. 5504, (Berlin, Heidelberg), pp. 288–302, Springer Berlin Heidelberg, 2009.

[6] P. Malacaria, “Algebraic foundations for quantitative information flow,” *Mathematical Structures in Computer Science*, vol. 25, pp. 404–428, 2014.

[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, (New York, NY, USA), pp. 73–85, ACM, 2009.

[8] G. Smith, “Recent Developments in Quantitative Information Flow (Invited Tutorial),” in *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pp. 23–31, jul 2015.

[9] 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.

[10] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, "Symbolic Quantitative Information Flow," *SIGSOFT Softw. Eng. Notes*, vol. 37, pp. 1–5, nov 2012.

[11] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. D;Amorim, "Quantifying Information Leaks Using Reliability Analysis," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, (New York, NY, USA), pp. 105–108, ACM, 2014.

[12] T. Chothia, Y. Kawamoto, and C. Novakovic, "LeakWatch: Estimating information leakage from java programs," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (M. Kutyłowski and J. Vaidya, eds.), vol. 8713 LNCS, (Cham), pp. 219–236, Springer International Publishing, 2014.

[13] C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu, "Precisely Measuring Quantitative Information Flow: 10K Lines of Code and Beyond," in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 31–46, mar 2016.

[14] S. McCamant and M. D. Ernst, "Quantitative Information Flow as Network Flow Capacity," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 43, 2008.

[15] C. Mu, *Computational Program Dependence Graph and Its Application to Information Flow Security.* Newcastle University, Computing Science, 2011.

[16] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, jul 1948.

[17] D. E. Denning, "A Lattice Model of Secure Information Flow," *Communications of the ACM*, vol. 19, pp. 236–243, may 1976.

[18] S. Muchnick, *Advanced compiler design implementation.* Morgan kaufmann, 1997.

[19] J. Graf, *Information Flow Control with System Dependence Graphs – Improving Modularity, Scalability and Precision for Object Oriented Languages.* PhD thesis, Karlsruher Institut für Technologie, nov 2016.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, pp. 319–349, jul 1987.

[21] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, "The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-driven Interpretation of Imperative Languages," in *Proceedings of the ACM SIGPLAN*

1990 Conference on Programming Language Design and Implementation, PLDI '90, (New York, NY, USA), pp. 257–271, ACM, 1990.

[22] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, pp. 26—-60, jan 1990.

[23] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 2, pp. 181–210, 1991.

[24] C. Click and K. D. Cooper, "Combining analyses, combining optimizations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 2, pp. 181–196, 1995.

[25] J. von Neumann, "First draft of a report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[26] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian journal of Mathematics*, vol. 8, no. 3, pp. 399–404, 1956.

[27] A.-H. Esfahanian, "Connectivity algorithms," 2013.

[28] S. Even, *Graph Algorithms.* New York, NY, USA: Cambridge University Press, 2nd ed., 2012.

[29] R. L. Rivest, T. H. Cormen, and C. E. Leiserson, *Introduction to Algorithms.* New York, NY, USA: McGraw-Hill, Inc., 1st ed., 1990.

[30] B. Wegbreit, "Property extraction in well-founded property sets," *IEEE Transactions on software engineering*, no. 3, pp. 270–285, 1975.

[31] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," in *In Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, pp. 108–120, 2000.

[32] L. Dadda, "Some schemes for parallel multipliers," *Alta frequenza*, vol. 34, no. 5, pp. 349–356, 1965.

[33] F. E. Allen, "Control flow analysis," in *Measuring Insecurity of Programs*, vol. 5, (New York, NY), pp. 1–19, ACM, 1970.

[34] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java.* New York, NY, USA: Cambridge University Press, 2nd ed., 2003.

[35] S. S. Muchnick and N. D. Jones, *Program flow analysis - theory and applications.* Prentice Hall Professional Technical Reference, 1981.

[36] M. S. Alvim, A. Scedrov, and F. B. Schneider, “When Not All Bits Are Equal: Worth-Based Information Flow,” in *Principles of Security and Trust* (M. Abadi and S. Kremer, eds.), (Berlin, Heidelberg), pp. 120–139, Springer Berlin Heidelberg, 2014.

[37] S. Chakraborty, K. S. Meel, and M. Y. Vardi, “Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls,” tech. rep., 2016.

[38] 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), pp. 1:1—-1:12, ACM, 2011.

[39] M. Backes, B. Köpf, and A. Rybalchenko, “Automatic Discovery and Quantification of Information Leaks,” in *2009 30th IEEE Symposium on Security and Privacy*, SP ’09, (Washington, DC, USA), pp. 141–153, IEEE, may 2009.

[40] J. Bechberger, “Besser Benchmarken,” Master’s thesis, Karlsruher Institut für Technologie, 2016.

[41] E. Zarpas, “Benchmarking SAT Solvers for Bounded Model Checking,” in *Theory and Applications of Satisfiability Testing* (F. Bacchus and T. Walsh, eds.), (Berlin, Heidelberg), pp. 340–354, Springer Berlin Heidelberg, 2005.

[42] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st ed., 2014.

[43] T. S. Heinze and W. Amme, “Sparse Analysis of Variable Path Predicates Based upon SSA-Form,” in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques* (T. Margaria and B. Steffen, eds.), (Cham), pp. 227–242, Springer International Publishing, 2016.

[44] M. Stephenson, J. Babb, and S. Amarasinghe, “Bitwidth Analysis with Application to Silicon Compilation,” vol. 35, 2002.

[45] Free Software Foundation, “Using the GNU Compiler Collection (GCC): Optimize Options.”

[46] S. Kim and S. McCamant, “Bit-Vector Model Counting Using Statistical Estimation,” in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Beyer and M. Huisman, eds.), (Cham), pp. 133–151, Springer International Publishing, 2018.

[47] L. de Moura, H. Rueß, and M. Sorea, “Bounded Model Checking and Induction: From Refutation to Verification,” in *Computer Aided Verification* (W. A. Hunt and F. Somenzi, eds.), (Berlin, Heidelberg), pp. 14–26, Springer Berlin Heidelberg, 2003.

[48] A. Komuravelli, A. Gurfinkel, and S. Chaki, “SMT-based model checking for recursive programs,” *Formal Methods in System Design*, vol. 48, pp. 175–205, jun 2016.

[49] A. Gurfinkel, T. Kahsai, and J. A. Navas, “SeaHorn: A Framework for Verifying C Programs (Competition Contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems* (C. Baier and C. Tinelli, eds.), (Berlin, Heidelberg), pp. 447–450, Springer Berlin Heidelberg, 2015.

[50] B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki, “Local Search Algorithms for Partial MAXSAT,” in *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI’97/IAAI’97, pp. 263–268, AAAI Press, 1997.

[51] M. Koshimura, “QMaxSAT : A Partial Max-SAT Solver system description,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 8, pp. 95–100, 2012.

[52] C. Mu, “Quantitative Information Flow for Security : A Survey Technical Report : TR-08-06 (updated on 2010),” 2010.

[53] J. Heusser and P. Malacaria, “Measuring Insecurity of Programs,” tech. rep., 2007.

[54] C. Mu and D. Clark, “Quantitative analysis of secure information flow via probabilistic semantics,” in *International Conference on Availability, Reliability and Security, 2009. ARES’09*, pp. 49–57, IEEE, 2009.

[55] V. Klebanov, N. Manthey, and C. Muise, “SAT-Based Analysis and Quantification of Information Flow in Programs,” in *Quantitative Evaluation of Systems* (K. Joshi, M. Siegle, M. Stoelinga, and P. R. D’Argenio, eds.), (Berlin, Heidelberg), pp. 177–192, Springer Berlin Heidelberg, 2013.

[56] F. Biondi, A. Legay, L.-M. Traonouez, and A. Wasowski, “QUAIL: A Quantitative Security Analyzer for Imperative Code,” in *Computer Aided Verification* (N. Sharygina and H. Veith, eds.), (Berlin, Heidelberg), pp. 702–707, Springer Berlin Heidelberg, 2013.

[57] Q.-S. Phan and P. Malacaria, “Abstract Model Counting: A Novel Approach for Quantification of Information Leaks,” in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS ’14, (New York, NY, USA), pp. 283–292, ACM, 2014.

[58] R. Chadha, U. Mathur, and S. Schwoon, “Computing Information Flow Using Symbolic Model-Checking,” in *34th Conference on Foundations of Software Technology and Theoretical Computer Science*, 2014.

[59] V. Klebanov, “Precise quantitative information flow analysis— a symbolic approach,” *Theoretical Computer Science*, vol. 538, pp. 124–139, 2014.

[60] D. Spagnuelo, C. Bartolini, and G. Lenzini, “Modelling metrics for transparency in medical systems,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (G. Livraga, V. Torra, A. Aldini, F. Martinelli, and N. Suri, eds.), vol. 10442 LNCS, (Cham), pp. 81–95, Springer International Publishing, 2017.

[61] M. Assaf, J. Signoles, E. Totel, and F. Tronel, “The Cardinal Abstraction for Quantitative Information Flow,” in *Workshop on Foundations of Computer Security 2016 (FCS 2016)*, (Lisbon, Portugal), jun 2016.

[62] Y. Kawamoto, F. Biondi, and A. Legay, “Hybrid Statistical Estimation of Mutual Information for Quantifying Information Flow,” in *FM 2016: Formal Methods* (J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, eds.), (Cham), pp. 406–425, Springer International Publishing, 2016.

[63] T. M. Ngo and Q. T. Duong, “A Tool to Compute the Leakage of Multi-threaded Programs,” in *Modern Approaches for Intelligent Information and Database Systems* (A. Sieminski, A. Kozierkiewicz, M. Nunez, and Q. T. Ha, eds.), (Cham), pp. 527–537, Springer International Publishing, 2018.

[64] I. Sweet, J. M. C. Trilla, C. Scherrer, M. Hicks, and S. Magill, “What’s the Over/Under? Probabilistic Bounds on Information Leakage,” in *Principles of Security and Trust* (L. Bauer and R. Küsters, eds.), (Cham), pp. 3–27, Springer International Publishing, 2018.

Erklärung

Hiermit erkläre ich, Johannes Bechberger, 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. Partial MAXSAT Based Leakage Computation

This chapter describes the transformation of the leakage approximation problem from Section 4.2 into a Partial MAXSAT (PMSAT) optimisation problem. It is more precise than the previously proposed minimum-vertex-cut based version, as it can take the *alternative* mapping into account. This mapping stores for each bit that a bit b depends on, the other bit that b can depend on instead of the original bit if there is such an alternative bit. The usage of the *alternative* mapping increases the precision of the approximation by taking all bits in the bit dependency graph into account when deciding for one of the two alternatives, instead of using a basic heuristic.

An instance of PMSAT consists of formulas in conjunctive normal form (CNF) that are separated into hard and soft constraints that work with the same set of variables. Formulas in CNF form consists clauses connected by logical and-operators, these clauses consist of variables, negated or not, that are connected by logical or-operators. A solution for a PMSAT instance is an instantiation of the variables in the formulas, so that all hard constraints and as many soft constraints as possible are fulfilled [50]. There are many solvers, but finding an optimal solution is NP-complete. The transformation starts by generating the hard constraints which ensure the soundness of the solution. Each bit b is associated with three boolean variables:

r_b If true, consider the dependencies of the bit instead of the bit itself for fixation.

c_b If true, fix the bit b.

d_b If true, require the consideration of either b or the bits b depends on.

The hard constraints for every statically unknown non-low-input bit b are given in the following. Low input bits have no hard constraints, as they are not considered in the leakage computation. Whenever a bit b is considered, then either the bit is really considered as fixed in the leakage computation or its dependencies are considered:

$$d_b \to (c_b \vee r_b)$$

If the dependencies are considered then all bits that b directly depends on are considered. This results in the following formula, assuming that b depends on $\alpha_1, \ldots, \alpha_m$ and that the alternative to considering the dependency α^i is β^i if there is no alternative, then $\alpha^i = \beta^i$:

$$r_b \to \wedge_{i=1}^{m}(d_{\alpha^i} \vee d_{\beta^i})$$

Converting these constraints into CNF is trivial.

d_{o^i} is added as a hard constraint for every output bit o^i, to ensure that all low output bits, or their dependencies, are considered. For all bits β with $weight(\beta) = \infty$, the constraint $d_\beta \rightarrow r_\beta$ is added, to ensure that these bits are not considered themself.

The soft constraint added for each statically unknown bit is $\neg c_b$. The number of c_b variables set to true in the solution is the approximated leakage.

Another problem, besides the complexity of solving the PMSAT instance, is that PMSAT solvers are optimised for instances that consist of few hard constraints and many soft constraints [50]. This is not the case here, as there is only one soft constraint and multiple hard constraints per bit.

Example The last part of this chapter shows how to analyse a basic example program given in Listing A.1 using PMSAT based leakage approximation. We assume single bit values.

```
if (h == 0) {
  o1 = 0b0
} else {
  o2 = 0b1
}
o = ϕ(o1, o2)
```

Listing A.1: Basic example with one bit values

This results in the following bits, as explained before in Section 4.1.4:

$$\begin{aligned}
(\mathtt{h} == 0)^0 &= (\hat{u},\ \{h^0\}) \\
o1^0 &= (\hat{0}, \varnothing) \\
o2^0 &= (\hat{1}, \varnothing) \\
o^0 &= (\hat{u},\ \{(\mathtt{h} == 0)^0\})
\end{aligned}$$

The transformation explained above generates the presented hard constraints for these bits:

$$\begin{aligned}
d_{h^0} &\rightarrow (c_{h^0} \lor r_{h^0}) \\
r_{h^0} &\rightarrow \mathit{true} \\
d_{(\mathtt{h}==0)^0} &\rightarrow (c_{(\mathtt{h}==0)^0} \lor r_{(\mathtt{h}==0)^0}) \\
r_{(\mathtt{h}==0)^0} &\rightarrow (d_{h^0} \lor d_{h^0}) \\
d_{o^0} &\rightarrow (c_{o^0} \lor r_{o^0}) \\
r_{o^0} &\rightarrow (d_{(\mathtt{h}==0)^0} \lor d_{(\mathtt{h}==0)^0}) \\
d_{o^0}
\end{aligned}$$

The soft constraints are:

$$
\begin{aligned}
&\neg c_{h^0}\\
&\neg c_{o^0}\\
&\neg(\texttt{h} == \texttt{0})^0
\end{aligned}
$$

The transformation into CNF is omitted for brevity. A PMSAT solver like *QMaxSAT* [51] solves this optimisation problem in less than one millisecond. The found optimal variable setting results in the expected leakage of one bit.

B. Tool Survey

This chapter presents a survey of the available tools in Quantitative Information Flow Control. The aim is to give an overview over these tools with their main characteristics. This survey focusses on tools that can analyse programs with inputs and outputs, excluding tools that analyse side-channels or database queries. The survey results are presented in Table B.1. The only other survey, to our knowledge, was conducted by Mu in 2010 [52].

Collection method We collected the tools by examining all papers that Google Scholar[1] returned for the search query `analysis evaluation tool "quantitative information flow" program`. This query includes all papers that contain the words `analysis` and `tool`, as the goal was to find analysis tools, and the words `evaluation` and `program`, as the goal was to find evaluated analyses. Furthermore the query part `"quantitative information flow"` ensures that all papers are related to Quantitative Information Flow.

The papers found by Google Scholar were then filtered manually to include only papers that contain the word `quantitative` in their abstract. Of these papers, only the papers that present an implemented analysis were finally considered. An implemented analysis is an analysis from a paper that presents an evaluation and mentions the implementation of a tool. The list of finally selected papers contains only one published paper per tool, as some tools were mentioned in multiple papers.

Validation The selected papers, found using Google Scholar, included all relevant tool papers that were known a-priori.

Limitations Due to the number of papers to survey, Google Scholar returned 514, it was not possible to examine all papers in detail. Relevant papers might omit the required keywords given before. The summaries of the papers given in Table B.1 are furthermore rather informal, as no attempt was made to fully understand the papers and their concepts.

Results The results, given in table B.1, show that the majority of the 20 analyses for Quantitative Information Flow are based on model counting and try to support a real-world language, but that no tool can practically soundly analyse programs with unbounded finite recursion and loops.

[1]https://scholar.google.com visited on the 20th and 21st of November 2018

Paper	Year	Category[1]	Technique[2]	Supported language[3]	Tool
[53]	2007	combination	semantics and observation	while	-
[14]	2008	dynamic	tainting	real-world	Flowcheck
[39]	2009	static	model counting	bounded while	-
[7]	2009	combination	model counting with tainting	real-world	-
[54]	2009	static	denotational semantics and distributions	while	-
[10]	2012	static	symbolic execution and bit-pattern model counting	if-statements	jpf-qif[4]
[55]	2013	static[5]	model counting with SAT models	real-world	-
[56]	2013	static	path counting on a markov chain model	restricted while	Quail
[11]	2014	static	model counting on paths	if-statements	jpf-qilura[4]
[12]	2014	dynamic	observation of distributions	real-world	Leakwatch
[57]	2014	static[5]	model counting with SMT Models	real-world	-
[58]	2014	static[5]	model counting with boolean programs	bounded while and functions	Moped-QLeak
[59]	2014	static	symbolic exec. and counting with verification conditions	while	-
[60]	2016	static[5]	model counting with SAT models	real-world	SharpPI
[61]	2016	static	denotational semantics with cardinal abstraction	while	Fhamator
[62]	2016	combination	observation and path counting	restricted while	HyLeak
[13]	2016	prob. static	symbolic execution with SAT models	real-world	Kite
[9]	2017	prob. static	model counting with SAT models	real-world	ApproxFlow
[63]	2018	static	symbolic execution with traces	if-statements, multi-threaded	-
[64]	2018	static	symbolic execution and sampling of distributions	while	TAMBA

Table B.1.: Quantitative Information Flow analysis tools for programs and their papers

[1] Analysis category, as described in Chapter 1, "combination" labels tools that use dynamic and static techniques

[2] Short summary of the main technique presented in the paper and used in the tool

[3] Language that a tool supports: "real-world" means the support of a real-world language like C or Java, "while" the support of a while-language, "if-statements" the support of a loop-free while-language

[4] The tool requires libraries that are not publicly accessible

[5] The tool only works theoretically for unbounded finite loops and recursion. In practice, only a limited recursion depth and a limited number of loop iterations can be analysed due to resource limitations.