Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

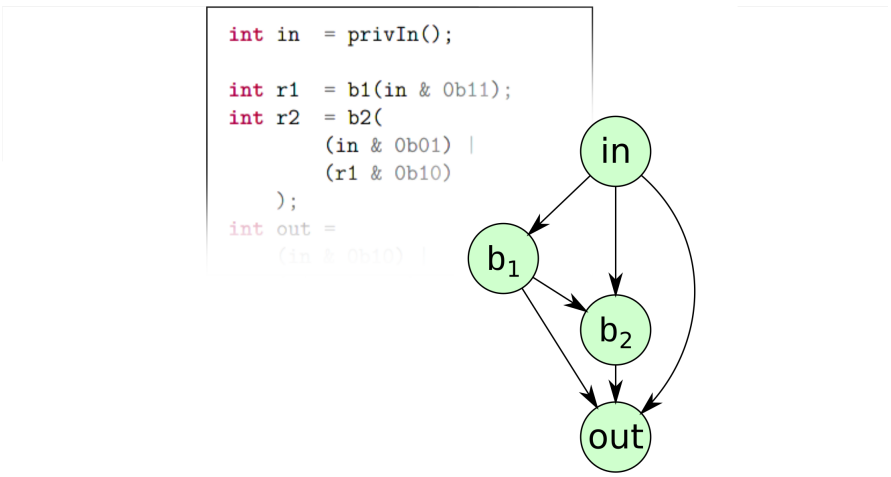
Lehrstuhl Prof. Dr.-Ing. Snelting

# Model Counting Based Quantitative Information Flow for Unbounded Loops and Recursions

Bachelorarbeit von

**Yannick Urbach**

an der Fakultät für Informatik




| | |
|---|---|
| **Erstgutachter:** | Prof. Dr.-Ing. Gregor Snelting |
| **Zweitgutachter:** | Prof. Dr. rer. nat. Bernhard Beckert |
| **Betreuende Mitarbeiter:** | M. Sc. Johannes Bechberger, M. Sc. Simon Bischof |
| **Abgabedatum:** | 20. April 2021 |

# Zusammenfassung

Quantitativer Informationsfluss ist das Problem, die Menge an privater Information, die von einem Programm über seine Ausgaben preisgegeben wird, zu bestimmen. Bounded Model Checking kann in Verbindung mit Model Counting für präzise Analysen des quantitativen Informationsflusses benutzt werden. Ein inherentes Problem dieses Ansatzes ist allerdings, dass Schleifen und Rekursionen beschränkt sein müssen. Wir stellen eine neuartige Analyse basierend auf Model Counting vor, die diese Einschränkung durch besondere Behandlung von Schleifen mit vielen Iterationen sowie tiefen Rekursionen abschwächt. Wir behandeln abgebrochene Iterationen und rekursive Aufrufe als Blackboxen und überapproximieren den Informationsfluss durch diese Blackboxen. Unsere Analyse unterstützt außerdem volle Ein- und Ausgabestreams, während vergleichbare Analysen typischerweise auf einem einfacheren Ein- und Ausgabemodell arbeiten.

Quantitative information flow is the problem of determining the amount of private information that a program leaks to its outputs. Bounded model checking in combination with model counting can be used for precise analyses of quantitative information flow. However, an inherent limitation of this approach is that loops and recursions must be bounded. We present a novel analysis based on model counting that alleviates this limitation by introducing a fallback for loops with many iterations and deep recursions. We treat aborted loop iterations and recursive calls as black boxes and overapproximate the information flow through them. Furthermore, our analysis supports full input and output streams, whereas comparable analyses typically use a simpler input and output model.

# Contents

# 1. Introduction

Programs often work on private information such as passwords, personal information of users, or industrial secrets. As processing that information is part of their purpose, they typically behave differently for different private information. This can be a problem if the difference in behavior affects publicly visible outputs: an attacker may use the information gained from observing the outputs to draw conclusions about the private information. Consider the password check in figure 1.1: The program notifies the user on whether they entered the correct password. While this is generally necessary in a password check, it reveals some information about the password. The discipline of secure information flow tackles the problem of identifying such leaks of private information. Two main questions may be asked with regards to secure information flow:

```
long password = privIn();
long input = pubIn();

boolean isCorrect = password == input;

pubOut(isCorrect);
```

**Figure 1.1.:** A simple password check.

**Does the program leak *any* private information?** Answering this question is referred to as *qualitative information flow analysis*. A number of analyses for qualitative information flow exist, and there are tools like JOANA [1] that are capable of analyzing large real-world programs.

Qualitative information flow analysis is useful when no leakage is acceptable. However, avoiding all leakage can be very challenging in some cases, and impossible or not desirable in others, such as in the password check. For another example, consider a checksum, like the parity

```
long in = privIn();
boolean parity = false;

for (int j = 0; j < 64; ++j) {
    parity ^= (in & (1 << j)) != 0;
}

pubOut(parity);
```

**Figure 1.2.:** Calculating the parity bit for a 64 bit integer.

bit calculated in figure 1.2. Observing the checksum reveals some information about the input, as it rules out all inputs that have a different checksum. However, it is impossible to guess the input from the checksum due to the small amount of information contained in the checksum. Nonetheless, it is considered a leak for qualitative information flow. Accepting that in some cases, leaks can not be avoided, leads to the following second question:

***How much* private information does the program leak?** Answering this more difficult question is referred to as *quantitative information flow analysis*. In the above example, only one bit of information about the input is leaked. Considering that the input consists of 64 bits, this may well be acceptable.

The analyses available for quantitative information flow are much more limited. One promising approach is to use bounded model checking in combination with approximate model counting to determine an estimate for the amount of information leaked. However, this approach is limited with regards to the programs which it can analyze. Bounded model checking can only handle a limited number of execution paths. Consequently, loops and recursions can only be analyzed up to a certain depth. This limitation is typically inherited by information flow analyses based on bounded model checking.

## 1.1. Contributions

In this thesis, we present a novel analysis that is based on bounded model checking and model counting, but avoids the limitation mentioned above. In our analysis, when loops exceed the limit to the number of iterations, the remaining iterations are treated as a black box. Similarly, when recursion exceeds the depth limit, the remaining recursive calls are treated as black boxes. We use bounded model checking to determine partial information flows in the remaining parts of the program. We then overapproximate the total information flow by taking into account those partial flows and assuming maximum information flow through the black boxes.

Furthermore, our analysis uses a stream-based input and output model. Existing analyses typically use a function-like input and output model, where inputs are passed to the program at the start of the execution, and outputs are returned when the execution finishes. Our analysis uses output buffers to allow for input and output at any point during the execution. We use maximum model counting to support public input. Our analysis supports a subset of Java, including unbounded loops and recursion, as well as dynamic dispatch.

## 1.2. Related Work

Smith [2] describes the theoretical foundations of quantitative information flow, in particular the use of min-entropy as an entropy measure for the leakage of a program. Fremont [3] introduces maximum model counting and describes its use for quantitative information flow with public inputs. We use maximum model counting for this purpose in our analysis. Espinoza [4] describes min-entropy-based quantitative information flow for cascades, which is integral to our approach for handling unbounded loops and recursion.

**ApproxFlow** ApproxFlow [5] is an approximative analysis for deterministic C programs that is based on bounded model checking and approximate model counting. It uses CBMC [6] as a bounded model checker. ApproxFlow delegates loop and recursion handling to CBMC, which employs unrolling and inlining with limited depth. Consequently, it has the limitations discussed above: loops and recursions are supported only up to a limited depth; loops and recursions beyond that depth can lead to underapproximations. ApproxFlow analyses information flow through a C function, treating parameters as private input and the return value as public output. Additional private input can be read through a stream-like interface. It does not support output streams, nor public input.

**jpf-qif** jpf-qif [7] is an analysis for deterministic Java programs based on the Java Pathfinder model checking platform. It assumes loops and recursions are bounded. Like ApproxFlow, it analyses a function and interprets parameters as private inputs and the return value as output. It does not support input or output streams, nor public input.

**nildumu** nildumu [8] is an analysis for deterministic programs in a custom language. It supports unbounded loops and recursion. The approach has similarities to our approach for unbounded loops and recursion (especially the alternative one described in appendix A) in so far that it uses network flow principles on a dependency graph. However, the entire analysis is based on this principle, whereas ours uses it in combination with the model counting based approach. In nildumu, private inputs and public outputs are provided through global variables. Nildumu does not support input and output streams, nor public input.

**TAMBA** TAMBA [9] is an analysis for deterministic and probabilistic programs in a custom language. It supports unbounded loops, but not recursion. Input and

output are function-like, with no support for streams or public input. TAMBA is based on the analysis described in [10], but extends it using symbolic execution in combination with model counting, as well as sampling.

# 2. Foundations

In this chapter, we discuss our program and attacker model, as well as the theoretical foundations of our analysis.

## 2.1. Program Model

In this thesis, we discuss deterministic programs that interact with their environment through input and output streams and, unless otherwise noted, terminate for every input. Programs may have several input and output streams. The sequence of input and output operations may depend on the inputs.

## 2.2. Attacker Model

We assume an attacker that interacts with a program with the goal of obtaining private information. We distinguish between public input streams and private input streams as well as public output streams and private output streams.

**Input and Output** Public input streams are controlled by the attacker: whenever the program reads a public input, the attacker may make the choice of input. In contrast, private input streams are not controlled by the attacker, and private input is invisible to the attacker. Similarly, public output is visible to the attacker, whereas private output is not. The attacker can observe the sequence of public input and output operations, including on which stream they occur, but not their timing.

The following is a formalization of the information that an attacker can observe according to our model:

**Observable Information** Let $S_I$ be the set of input streams, $S_O$ the set of output streams, $V$ the set of values, and $V(s) \subset V$ the set of values valid for a stream $s$. At any point during the execution, the information available to the attacker is a sequence

$$\textit{observable information} \in (S_I \cup \{(s, v) \mid s \in S_O, v \in V(s)\})^*.$$

The sequence is initially empty. A public input operation appends its stream to the sequence, a public output operation appends a tuple $(s, v)$ where $s$ is the output stream and $v$ is the value written to it.

We illustrate this formalization for the short example program in figure 2.1: The program has the input streams `pubIn1` and `pubIn2` and the output streams `pubOut1` and `pubOut2`. At the end of the program, the observable information is the sequence

$$[\mathit{pubIn1}, (\mathit{pubOut1}, a), (\mathit{pubOut2}, b), \mathit{pubIn2}].$$

```
pubIn1();
pubOut1(a);
pubOut2(b);
pubIn2();
```

**Figure 2.1.:** Simple example for input and output streams

We are interested in the amount of information about the private inputs that is exposed to the attacker through the execution of the program. We will refer to this quantity as *leakage*. In particular, the goal of this thesis is to provide a sound approximation of the leakage. By sound in this context, we mean that the approximation is an overapproximation of the actual leakage.

## 2.3. Quantitative Information Flow

Smith [2] proposes a measure for quantitative information flow. This measure is based on min-entropy, which in turn is defined using the concept of vulnerability. We reproduce the definitions for both of those concepts in the following:

**Definition 1** (Vulnerability)**.** Let $X$ be a random variable, and $\mathcal{X}$ the set of possible values of $X$. The *vulnerability* of $X$ is

$$V(X) = \max_{x \in \mathcal{X}} P[X = x].$$

$V(X)$ is the probability of guessing the value of $X$ correctly on the first attempt, maximized over all possible values of $X$.

**Definition 2** (Conditional Vulnerability)**.** Let $X$ and $Y$ be random variables, and $\mathcal{X}$ and $\mathcal{Y}$ their sets of possible values. The *conditional vulnerability* of $X \mid Y$ is

$$V(X \mid Y) = \sum_{y \in \mathcal{Y}} P[Y = y] V(X \mid Y = y)$$

Analogously to $V(X)$, $V(X \mid Y)$ is the probability of guessing the value of $X$ correctly on the first attempt, knowing the value of $Y$, maximized over all possible values of $X$.

**Definition 3** (Min-Entropy)**.** The *min-entropy* $V(X)$ and *conditional min-entropy* $V(X \mid Y)$ are given by

$$\mathrm{H}_\infty(X) = \log \frac{1}{V(X)},$$
$$\mathrm{H}_\infty(X \mid Y) = \log \frac{1}{V(X \mid Y)}.$$

Like Shannon-entropy, min-entropy can be interpreted as a measure for the degree of uncertainty about the value of a random variable. In fact, for uniformly distributed variables, Shannon-entropy and min-entropy are equal [11]. They differ in their behavior for non-uniformly distributed random variables. Shannon entropy is affected by all possible values, whereas min-entropy depends only on the most probable value. Consequently, min-entropy indicates a low degree of uncertainty whenever there is a highly probable value, even if there are many other possible values.

In Smith's model, private input and public output are modelled as random variables $H \in \mathcal{H}$ and $L \in \mathcal{L}$. A deterministic program defines a function $f : \mathcal{H} \to \mathcal{L}, H \mapsto L$.

Following the interpretation of min-entropy as degree of uncertainty, the *information leaked* by the program is defined as follows:

**Definition 4** (Information leaked)**.**

$$\begin{aligned} \textit{information leaked} &= \textit{initial uncertainty} - \textit{remaining uncertainty} \\ &= \mathrm{H}_\infty(H) \qquad\qquad - \mathrm{H}_\infty(H|L) \end{aligned}$$

In general, determining the information leaked requires determining the conditional probabilities of outputs of the program. However, Smith gives an upper bound for which this is not necessary [11]. For uniformly distributed private inputs, this upper bound is also the exact value.

**Theorem 1** (Upper bound to the information leaked)**.**

$$\textit{information leaked} \leq \log |\mathcal{L}|.$$

If the program is deterministic and $H$ is uniformly distributed, then

$$\textit{information leaked} = \log |\mathcal{L}|.$$

This upper bound depends only on the number of possible public outputs. As such, it is much easier to determine and can be used as a sound approximation.

A few additional steps are necessary to adapt these definitions and findings to our attacker model. We expand the approximation to allow for public inputs by taking the maximum over all possible public inputs:

**Theorem 2** (Information leaked for public input)**.** Let $\mathcal{I}$ be the set of possible public inputs. Let $\mathcal{L}_i$ be the set of outputs possible for the public input $i$. Then

$$\begin{aligned} \textit{information leaked} &\leq \max_{i \in \mathcal{I}} \log |\mathcal{L}_i| \\ &= \log \max_{i \in \mathcal{I}} |\mathcal{L}_i| \end{aligned}$$

According to our program model, public input and public output can be interleaved. This may affect leakage in two ways:

Firstly, the attacker can base their choice of public input on information previously obtained. However, by taking the maximum leakage over all possible public inputs, we already account for the choice of public input that results in the highest leakage.

Secondly, as discussed in section 2.2, the attacker may obtain information through the sequence of public input and output operations alone. Consider the example on the left side of figure 2.2: The output of this program is the same regardless of the private input. However, the attacker can observe whether the output is written *before* or *after* they are prompted for input. Following the formalization from section 2.2, if `b1 == true`, the attacker observes $[\mathit{pubIn}, (\mathit{pubOut}, 0)]$, otherwise they observe $[(\mathit{pubOut}, 0), \mathit{pubIn}]$. Consequently, they can deduce the value of `b1`.

**Without sequence output**

```
boolean b1 = privIn();
boolean b2;

if (b1) {
    b2 = pubIn();
    pubOut(0);
} else {
    pubOut(0);
    b2 = pubIn();
}
```

**With sequence output**

```
boolean b1 = privIn();
boolean b2;

if (b1) {
    b2 = pubIn();
    seqOut(1);

    pubOut(0);
    seqOut(2);
} else {
    pubOut(0);
    seqOut(2);

    b2 = pubIn();
    seqOut(1);
}
```

**Figure 2.2.:** Simple example for information being leaked through the sequence of input and output, and the simulation of this leakage through a sequence output stream.

To account for this leakage, we assign a unique number to every public input or output stream and introduce a special additional public output stream which we call the *sequence output* stream. Whenever an input or output operation occurs, we write the number of the stream to the sequence output stream. For the program in figure 2.2, the result of this is given on the right side.

The regular output remains the same, but the sequence output is either `[1, 2]` or `[2, 1]`, reflecting the information leaked through the sequence of input and output operations.

## 2.4. Bounded Model Checking

Bounded model checking is a technique for formal verification of programs [12]. The core principle of bounded model checking is to convert the program to a Boolean formula and then check properties on that formula. Because the size of a Boolean formula is finite, bounded model checking can only cover a finite number of execution paths, hence the name "*bounded* model checking".

Bounded model checkers represent the variables in the program using Boolean variables. Because the value of a variable may change throughout the execution of the program, it is generally not enough to introduce a single Boolean variable for each bit of a program variable. Instead, after every change of a program variable, a new set of Boolean variables is introduced to represent the value between that change and the next. Each such representation is referred to as an instance of the program variable.

Through systematic exploration of execution paths, a formula $\varphi$ on the Boolean variables is constructed. $\varphi$ is *true* for an interpretation $\sigma$ if and only if $\sigma$ translates to an interpretation of the program variables that can occur in an execution of the program. An interpretation $\sigma$ for which $\varphi$ is *true* is known as a model of $\varphi$. Consider the example in figure 2.3 for illustration:

We use the Boolean variable $A$ to represent `a`, $B_1$ to represent `b` before the `if` statement, and $B_2$ to represent `b` after the `if` statement. `a` is *non-determinate*, meaning it can be either `true` or `false`, rather than being assigned a specific value. Non-determinate values allow specification of programs that depend on inputs or randomization. The program can be represented by the following formula in conjunctive normal form:

```
boolean a = nondet;
boolean b = true;

if (a) { b = false; }
```

**Figure 2.3.:** Example program for model checking.

$$\varphi(A, B_1, B_2) = B_1 \wedge (A \vee B_2) \wedge (\neg A \vee \neg B_2)$$

$\varphi$ has the models $(true, true, false)$ and $(false, true, true)$.

In bounded model checking, $\varphi$ can be used to verify properties of the program. For example, to verify that at the end of the execution of the above program, `a != b`, we check if there is a model of $\varphi$ with $A = B_2$, which is not the case. For more complex cases, properties can be formulated as instances of the Boolean satisfiability problem SAT, which can be solved automatically using a SAT solver.

For our purposes, we make use of $\varphi$ in a different but related way: To determine the amount of information leaked by a program, we need to find the number of possible outputs. Regarding $\varphi$, this translates to counting the number of interpretations for the corresponding Boolean variables that can be expanded to a model of $\varphi$. This problem is known as the projected model counting problem #∃SAT.

## 2.5. SAT Model Counting

In this section, we introduce the projected model counting problem #∃SAT, as well as an extension thereof, the maximum model counting problem Max#SAT. For context, we begin with the model counting problem #SAT before successively extending it to #∃SAT and Max#SAT. The model counting problem #SAT can be defined as follows [13]:

**Definition 5** (#SAT)**.** Let $\varphi(X)$ be a Boolean formula over a set of variables $X$. The model counting problem #SAT is the problem of determining

$$|\{X \mid \varphi(X)\}|.$$

Informally, this is the number of different interpretations for $X$ such that $\varphi(X)$ evaluates to *true*.

An extension of #SAT known as the projected model counting problem or #∃SAT introduces a sampling set and restricts the count to the variables in the sampling set [14]:

**Definition 6** (#∃SAT)**.** Let $\varphi(X, Y)$ be a Boolean formula over sets of variables $X$ and $Y$. The projected model counting problem #∃SAT with sampling set $X$ is the problem of determining

$$|\{X \mid \exists Y : \varphi(X, Y)\}|,$$

or informally, the number of different interpretations for $X$ that can be expanded to a model of $\varphi$.

For programs without public input, the problem of determining the amount of information leaked directly translates to #∃SAT on the formula constructed by a bounded model checker, with the sampling set consisting of the output variables. For programs with public input, we need to maximize the model count over all possible public inputs, which leads to a problem known as Max#SAT [3].

**Definition 7** (Max#SAT)**.** Let $\varphi(X, Y, Z)$ be a Boolean formula over sets of variables $X$, $Y$ and $Z$, and $\mathcal{X}$ the set of interpretations for $X$. Max#SAT is the problem of determining

$$\max_{x \in \mathcal{X}} |\{Y \mid \exists Z : \varphi(x, Y, Z)\}|.$$

Informally, Max#SAT is #∃SAT with sampling set $Y$, maximized over the variables in the maximization set $X$.

To determine the information leaked by a program with public inputs, we use the public outputs as the sampling set, and the public inputs as the maximization set.

For both #∃SAT and Max#SAT, "probably approximately correct" model counters exist [15, 16, 3]. They allow specification of a tolerance $\epsilon$ and a confidence $1 - \delta$, and guarantee that

$$P[c_e/(1+\epsilon) \leq c_a \leq c_e(1+\epsilon)] \geq 1 - \delta,$$

where $c_e$ is the exact count and $c_a$ is the approximated count. In particular, this means that

$$P[c_e \leq c_a(1+\epsilon)] \geq 1 - \delta.$$

Consequently, $c_a(1+\epsilon)$ can be used for a "probably sound" approximation for the leakage, i.e. one that is sound with a probability of at least $1 - \delta$. As the approximation for the leakage is logarithmic in the model count, this $\epsilon$-adjustment translates to a constant offset:

$$\log(c_a(1+\epsilon)) = \log(c_a) + \log(1+\epsilon)$$

# 3. Design

In the previous chapter, we discussed the foundations of quantitative information flow analysis using bounded model checking and model counting. In this chapter, we discuss how we extend this principle to provide the features that set our analysis apart from existing analyses. In particular, we discuss output streams and unbounded loops and recursion.

## 3.1. Output Streams

In our program model in section 2.1, we chose a stream-based input and output model. We believe that this reflects many real-world applications more accurately than a simpler model. However, our general model counting based approach assumes a single output statement at the end of the execution. Additional considerations are therefore necessary to implement the stream-based output model.

```
boolean in = privIn();

if (in)
    pubOut(false);

pubOut(in);

if (!in)
    pubOut(true);
```

**Figure 3.1.:** An example program with interleaved input and output

Throughout this section, we will use the following example in figure 3.1 for illustration. This program has three output statements, each of which depends on the input. Still, the program is considered not to leak any information. An observer would see the sequence `[false, true]` on the output stream, regardless of the input.

As discussed in section 2.3, we introduce the special sequence output stream. The

input and output statements are replaced follows, by adding an additional output statement for the sequence output stream:

```
pubIn<a>();          →          pubIn<a>();
                                seqOut(<a>);

pubOut<a>(<v>);      →          pubOut<a>(<v>);
                                seqOut(<a>);
```

For the example in figure 3.1, this is demonstrated in figure 3.2.

```
boolean in = privIn();

if (in) {
    pubOut(false);
    seqOut(OUT_1);
}

pubOut(in);
seqOut(OUT_1);

if (!in) {
    pubOut(true);
    seqOut(OUT_1);
}
```

**Figure 3.2.:** The example from figure 3.1 with the sequence output stream `seqOut` added. `OUT_1` is the stream number of the output stream `pubOut`.

By doing so, we cover the information leaked through the sequence in which input and output statements occur. What remains is combining the values of the output statements into a single output. We will discuss two approaches to achieve this.

### 3.1.1. Modelled as Separate Variables

We will first discuss a naive approach: Each output statement shall be treated as a separate output variable. We assume that the program is already unwound and inlined as described in section 3.2, such that every output statement is reached at most once. To model the sequence output, we use a Boolean variable for each input and output statement that indicates whether the statement was reached.

Schematically, the input and output statements are replaced as follows, where <i> is a unique index of each output statement:

```
pubIn<a>();          →          pubIn<a>();
                                seqOut<i> = true;

pubOut<a>(<v>);      →          pubOut<a>_<i> = <v>;
                                seqOut<i> = true;
```

```
boolean in = privIn();
boolean pubOut0, pubOut1, pubOut2;
boolean seqOut0, seqOut1, seqOut2;

if (in) {
    pubOut0 = false;
    seqOut0 = true;
}

pubOut1 = in;
seqOut1 = true;

if (!in) {
    pubOut2 = true;
    seqOut2 = true;
}
```

**Figure 3.3.:** The example from figure 3.1 modelled using separate variables

Figure 3.3 demonstrates this for the example from figure 3.1. In this modification of the program, knowing `pubOut1` is enough to determine the full input, resulting in a leakage of one bit. While this is not the correct result with regards to the initial program, it can never be *lower* than the actual leakage. We can compute all information available to the attacker according to our attacker model from the values of the output variables: For each of the original public input and output statements we append the appropriate value to the sequence if the corresponding sequence output variable is set, and do nothing otherwise. For the above program, this would result in the sequence $[(0, false), (0, true)]$ as expected. This means that this naive approach can still serve as a sound approximation for the actual leakage. However, depending on the program, the approximation can be much higher than the actual leakage.

### 3.1.2. Modelled Using an Output Buffer

One approach to correctly model output streams is to use an output buffer and an offset into that buffer. An output statement, both on regular output streams and the sequence output stream, is translated into writing its output into the buffer cell specified by the offset and then incrementing that offset.

Schematically, the input and output statements are replaced as follows:

```
pubIn<a>();          →          pubIn<a>();
                                seqOut[seqOutOff] = <a>;
                                seqOutOff++;

pubOut<a>(<v>);      →          pubOut<a>[pubOutOff<a>] = <v>;
                                pubOutOff<a>++;
                                seqOut[seqOutOff] = <a>;
                                seqOutOff++;
```

Figure 3.4 demonstrates this for the example from figure 3.1. By now treating the output buffers as output, one gets the expected result: a constant output of `[false, true]`.

A disadvantage of this buffer-based approach is that the buffer size must be set. For a correct result, the buffer size should be an upper bound to the number of actual outputs. A smaller buffer could lead to a result that is lower than the actual leakage. The total number of output statements can be used as buffer size, but that may be much higher than the actual maximum. However, it is not always easy to determine a smaller upper bound. Furthermore, this approach requires far more SAT variables, which increases the time required for model counting.

### 3.1.3. Hybrid Approach

It is possible to combine the two approaches outlined above as follows: As long as the offset is less than the size of the buffer, the buffer-based approach is used. If further outputs beyond the capacity of the buffer occur, then those outputs are treated as separate output variables. Using this approach, choosing a correct buffer size is not essential to get sound results, but one still benefits from exact results as long as the buffer size is not exceeded.

```
boolean in = privIn();

boolean[] pubOut = new boolean[3];
int pubOutOff = 0;

int[] seqOut = new int[3];
int seqOutOff = 0;

if (in) {
    pubOut[pubOutOff] = false;
    pubOutOff++;

    seqOut[seqOutOff] = OUT_1;
    seqOutOff++;
}

pubOut[pubOutOff] = in;
pubOutOff++;

seqOut[seqOutOff] = OUT_1;
seqOutOff++;

if (!in) {
    pubOut[pubOutOff] = true;
    pubOutOff++;

    seqOut[seqOutOff] = OUT_1;
    seqOutOff++;
}
```

**Figure 3.4.:** The example from figure 3.1 modelled using output buffers

## 3.2. Loops and Recursion

Loops and recursion are essential to many real-world programs. Unfortunately, they are a major challenge for quantitative information flow control. As discussed in section 2.4, bounded model checking only supports a finite number of execution paths. Because of that limitation, existing bounded model checkers typically employ unwinding and inlining with limited depth [6, 17, 18]. We will discuss both of these techniques before presenting an alternative that treats loops and recursions as black boxes to allow for unbounded loops and recursions.

### 3.2.1. Unwinding

Unwinding refers to transforming a loop to a finite repetition of its body. In doing so, loop conditions have to be changed as well, such that they control whether control flow progresses further along the sequence of copies of the loop body, rather than controlling a conditional backwards jump. Figure 3.5 illustrates how a simple loop can be unwound with three iterations.

**Before unwinding**

```
int i = 0;

while (i < 2) {
    body();
    ++i;
}
```

**After unwinding**

```
int i = 0;

if (i >= 2) goto end;
body();
++i;
if (i >= 2) goto end;
body();
++i;
if (i >= 2) goto end;
body();
++i;

end:
```

**Figure 3.5.:** Unwinding a simple loop

An obvious shortcoming of this approach is that a reasonable number of iterations has to be determined. Following Rice's theorem, this is not generally possible, and even in cases where it is possible, sufficiently unwinding the loop is often not viable. In particular, conditions that depend on inputs can easily render the approach unfeasible, as the example in figure 3.6 demonstrates. Unwinding such loops is often unacceptable for both runtime and memory requirements. Infinite loops can never accurately be implemented through unwinding.

```
long in = privIn();
long out = 0;

for (long l = 0; l < in; ++l)
    ++out;
```

**Figure 3.6.:** A loop depending on input. The loop will run for `in` many iterations. Because `in` is unknown, the loop has to be unwound with at least $2^{63}$ iterations to cover all possible cases.

### 3.2.2. Inlining

Inlining refers to replacing a function call with the body of the function. This serves a similar purpose for recursion as unwinding does for loops: up to a limited depth, the recursive function can be simulated without using recursion.

**Dynamic dispatch** For languages with dynamic dispatch, the dispatching must be recreated as well. This includes Java, as well as most other object-oriented languages. The dispatching can be simulated through a simple if-else-cascade that checks the type of the polymorphic variable and executes the applicable method implementation. Figure 3.7 demonstrates this for an example of an overridden method call.

**Example of dynamic dispatch**

```
class Parent {
    void foo() {
        /* behaviour 1 */
    }
}

class Child extends Parent {
    @Override
    void foo() {
        /* behaviour 2 */
    }
}

Parent p = ...;
p.foo();
```

**Inlined dispatch of this call**

```
if (p instanceof Parent) {
    // behaviour 1
} else if (p instanceof Child) {
    // behaviour 2
} else {
    // none of the possible classes
    // match, must be null
    throw new NullPointerException();
}
```

**Figure 3.7.:** Inlining a dynamically dispatched call

**Recursive functions** Recursive functions can not fully be inlined, as that would, due to their recursive nature, result in an infinitely long program. Analogously to loop unwinding, one can set a finite limit to the depth of the recursion to mitigate that. However, doing so introduces the same problems as loop unwinding: a reasonable limit to the depth has to be determined, and recursive functions whose actual depth depends on input are often unfeasible.

### 3.2.3. Black boxes

Despite the discussed shortcomings, unwinding and inlining remain essential techniques for dealing with loops and recursion. However, they are far from universally applicable solutions. To broaden the range of programs that can be covered by quantitative information flow analysis, we propose a technique for dealing with some of the situations in which they are unfeasible.

The goal of this technique is to isolate parts of the program that can not accurately be modeled by other techniques and treat them as "black boxes". We assume the worst case for the behaviour of the black boxes: that they behave in such way with regards to the variables on which they depend and which they modify, that the leakage of the program as a whole is maximized. We will refer to values on which a black box depends as the *input values* of that black box, and to values that depend on a black box as the *output values* of that black box.

Note that if a black box makes outputs, we have to assume it leaks all of its input values. Due to the sequence leakage described in section 2.2, this also applies to black boxes that read public input. Similarly, if a black box reads private input, we have to assume the output values are composed entirely of private input. A black box that both reads input and writes output can leak any amount of information, such that no sound approximation can be given. For simplicity, we omit these considerations in the analysis described below and instead impose the following restrictions on black boxes:

- Black boxes shall not make public outputs
- Black boxes shall not read inputs

Using the analysis as described for programs that violate these restrictions will likely lead to underapproximation.

In the following, let $p$ be a program that is fully unrolled and inlined except for a number of black box sections $\{b_1, b_2, \ldots, b_n\}$, labeled in order of their occurence in $p$. We want to construct a graph from those black box sections to represent the paths through which information may be leaked. To that end, we add two special nodes $b_0 = in$ and $b_{n+1} = out$ to represent private input and public output respectively. We then construct the following graph:

**Definition 8** (Black box Graph)**.** The black box graph $G$ for $p$ is given by

$$\begin{aligned} G &= (V, E), \\ V &= \{\underbrace{b_0}_{=in}, b_1, b_2, \ldots, b_n, \underbrace{b_{n+1}}_{=out}\}, \\ E &= \{(b_i, b_j) \mid b_i, b_j \in B, i < j\}. \end{aligned}$$

Intuitively, information can only flow to later black box sections, never to earlier ones. Additionally, information can flow from *in* to any black box section, and from any black box section to *out*, as well as directly from *in* to *out*. For a program with two black box sections, this graph is shown in figure 3.8.

Given this graph, a natural approach would be to determine the maximum possible information flow along its edges and treat the resulting weighted graph as a flow network. This would allow the use of regular maximum flow algorithms to find an approximation for the leakage. We discuss this approach in appendix A in more detail. However, as we argue there, the approximation provided by the following simpler approach is at least as low, and in some cases lower.

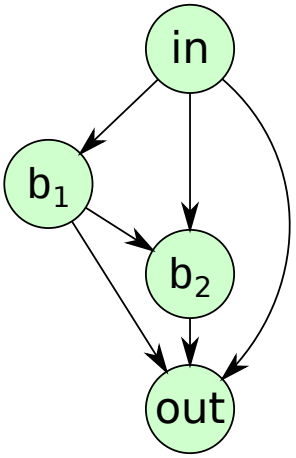



**Figure 3.8.:** Black box graph for a program with two black boxes

Instead of working on individual edges, we examine the information flow across the following cuts:

**Definition 9.** The *inter black box cuts* $C_i$ of $G$ are

$$C_i = (\{b_j \mid 0 \leq j \leq i\}, \{b_j \mid i < j \leq n+1\}).$$

We introduce the concept of *partial leakage* as a measure for the amount of information that flows across an *inter black box cut*:

**Definition 10.** The *partial leakage* $l_p(C_i)$ is the *leakage* of the program resulting from the following modification of $p$:

- The black box sections $B$ are omitted.

- For $j \leq i$, the output values of $b_j$ are replaced with private input.
- For $j > i$, the output values of $b_j$ are replaced with *unknown* values, and the input values of $b_j$ are considered public output.

*Unknown* values are treated like public input, but do not affect the sequence output stream.

Any information that is leaked by $p$ must pass through each *inter black box cut*. This means that the leakage of $p$ can not exceed any of the partial leakages $l_p(C_i)$ [4]. Consequently, $\min_{C_i} l_p(C_i)$ is a *sound* approximation for the leakage of $p$. Figure 3.9 demonstrates this approach for a simple program.

**Example program**

```
int in  = privIn();

int r1  = b1(in & 0b11);
int r2  = b2(
        (in & 0b01) |
        (r1 & 0b10)
    );
int out =
    (in & 0b10) |
    (r1 & ~0b10);

pubOut(out);
```

**Black box graph with cuts**

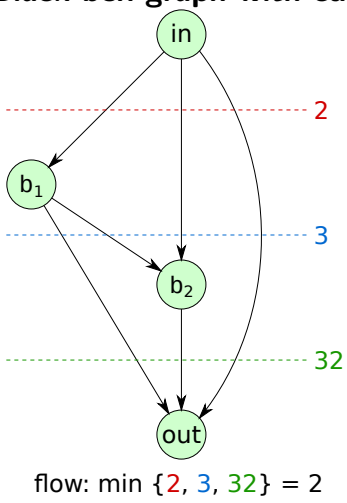



**Figure 3.9.:** Example for using partial leakage across cuts to determine a sound approximation; b1 and b2 are black boxes.

It is worthwhile to reflect on how this technique affects the probability of a sound approximation when using a probably approximately correct model counter. The leakage reported by this technique is a minimum over results of the model counter. The guarantees provided by the model counter are valid for each of the results. We assume that the results have already been adjusted to "probably sound" results as proposed in section 2.5. Taking the minimum over the results does not affect the tolerance. However, for the reported leakage to not be sound, it is enough that a single run of the model counter reports a leakage below the actual leakage. Consequently, the overall confidence is lower the more model counter runs are performed. This can be mitigated in two ways: The confidence setting passed to the model counter can be adjusted accordingly, but doing so significantly increases the runtime. Alternatively, the run that reported the lowest leakage can be repeated with a different seed. This

yields a result with the desired guarantees for the corresponding cut, which can be used as a probably sound approximation of the overall leakage with the desired confidence.

# 4. Implementation

The approach described in this thesis has been implemented in the tool approxflow-java[1]. In this chapter, we discuss implementational details of approxflow-java. Approxflow-java uses JBMC [17] as a bounded model checker. JBMC is based on CBMC but operates on Java Bytecode rather than C source code. As a model counter, ApproxMC [16, 19] or MaxCount [3] is used, depending on whether there are public inputs in the program. Figure 4.1 provides an overview over the architecture of approxflow-java. The part in the staggered box may be run multiple times for black box handling as described in section 3.2.3.

## 4.1. Target Language

Approxflow-java is intended for a subset of Java 8. Most basic language features are supported, including classes, methods, control flow statements, and all primitive types.

The following limitations apply:

- Programs must be single-threaded.
- Exception handling is not supported.
- Reflection is not supported.
- Inputs, outputs, and black box inputs and outputs must be primitively typed.
- Loops treated as black boxes must only operate on local variables.

[1] https://github.com/yannick-urbach/approxflow-java

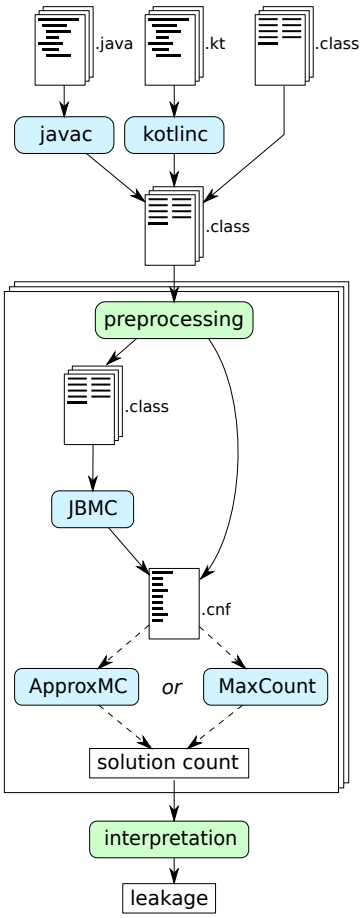



**Figure 4.1.:** Architecture of approxflow-java

## 4.2. Program Annotation

Approxflow-java uses a mostly annotation-based interface for specifying the additional information about the program that is required for the analysis.

An input stream is specified by annotating a method with `@PrivateInput` or `@PublicInput`. An output stream is specified by annotating a parameter with `@PublicOutput`.

The treatment of loops and methods, specifically unroll iterations, recursion depth for inlining, and black box handling, can be configured per method using annotations. Unrolling and inlining can be combined with black box handling as a fallback for remaining iterations and recursions.

JBMC provides a way of supplying additional constraints. In the context of quantitative information flow, this allows for example restriction of inputs, or manu-

ally improving the approximation provided by black box handling (section 3.2.3). Approxflow-java can optionally translate conventional Java assertions into such constraints. Additionally, object invariants can be specified as instance methods:

```
@Invariant
boolean aAlwaysPositive() { return a > 0; }
```

# 5. Evaluation

In this chapter, we evaluate approxflow-java with a number of test programs and compare the results with ApproxFlow where applicable.

**Limitations** Unfortunately, we encountered a problem where in some cases, MaxCount invariably returned a model count of zero when the number of variables in the sampling and maximization sets was too high. As the problem did not occur for equivalent programs with smaller input and output widths, we strongly suspect this problem lies with MaxCount, rather than our analysis. Nonetheless, it forced us to adapt some of the test programs that we intended to use for evaluation. For the Laundry examples in particular, this appears to defy the purpose of the test programs: For small amounts of data, fully unrolling and inlining the loops and recursions is viable in those cases when using ApproxFlow. However, as the adapted examples otherwise behave just like the original, we believe they are still fit to demonstrate our analysis.

## 5.1. Tested Programs

We briefly introduce the test programs that we used for evaluation. Some of the test programs were adapted from the case studies in [20], the remaining ones were developed by us. The code for the test programs can be found in appendix B.

**Parity** This is the parity example from chapter 1.

**Battlebits** In this interactive guessing game, a player attempts to hit set bits on a secret board, similarly to the board game *Battleship*. After every "shot", the program reports whether the shot was a "hit", i.e. whether the chosen bit is set.

**Laundry** This is a simple example of the problem of loop conditions that depend on inputs. As such, it demonstrates the usefulness of our black box approach compared to unwinding.

**Recursive Laundry** This is the recursive equivalent of the Laundry example.

**Partial Laundry** This variant of the laundry example only "launders" the first four bits. It demonstrates that the black box approach can in some cases return approximations that are much higher than the actual leakage.

**Voting 1** This is a variant of the single preference voting protocol from [20], which was simplified to match the limitations of our analysis. This variant models a simple for/against vote, such as in a referendum.

**Voting 2** Another simplified variant of the single preference voting protocol, with a higher number of options, but only three voters.

**Smart Grid** This is a variant of the smart grid case study from [20].

## 5.2. Evaluation Process

The evaluation was done on a desktop computer with an Intel Core i5-6600 CPU, 16GiB of RAM and a Samsung 970 Evo SSD.

**Approxflow-java** Approxflow-java was built and executed using OpenJDK version 1.8.0_282, and the same was used to compile the benchmark programs. We used JBMC version 5.25.0 and ApproxMC version 4.0.1. We used a modified version of MaxCount version 1.0.0 [21] that supports the newer versions of ApproxMC.

**ApproxFlow** We used a modified version of ApproxFlow [22] that allows for specifying the unroll depth. We used ApproxMC version 4.0.1 for ApproxFlow as well.

**Configuration** With regards to loops and recursion, we tested each tool in four configurations per test program. Three of those were with the unwinding and inlining limits 2, 8, and 32 respectively, and black box handling enabled for approxflow-java. The fourth configuration was with the lowest limit for each test program that fully unrolled and inlined the loops and recursions of the program, and with black box handling disabled for approxflow-java. *Battlebits* violates our restrictions on black boxes, and was therefore only tested with the fourth configuration.

For the model counters, we used the parameters $\epsilon = 0.8$ and $\delta = 0.2$ in both cases, as well as $k = 10$ for MaxCount.

## 5.3. Results

| Program | actual | approxflow-java | | | | ApproxFlow | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 2 | 8 | 32 | $\infty$ | 2 | 8 | 32 | $\infty$ |
| Parity | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| Battlebits | 3.0 | -[1] | -[1] | -[1] | 3.0 | -[2] | -[2] | -[2] | -[2] |
| Laundry | 7.0 | 7.0 | 7.0 | 7.0 | -[3] | 1.6 | 3.2 | 5.0 | 7.0 |
| Recursive Laundry | 7.0 | 7.0 | 7.0 | 7.0 | -[3] | 8.0 | 8.0 | 8.0 | 7.0 |
| Partial Laundry | 4.0 | 7.0 | 7.0 | 7.0 | 4.0 | 1.6 | 3.2 | 4.0 | 4.0 |
| Voting 1 | 5.0 | 8.0 | 8.0 | 5.0 | 5.0 | 1.6 | 3.2 | 5.0 | 5.0 |
| Voting 2 | 10.0 | 24.0 | 24.0 | 9.9 | 9.9 | 3.3 | 7.4 | 10.0 | 10.0 |
| Smart Grid | 5.0 | 16.0 | 15.9 | 5.0 | 5.0 | 1.6 | 3.7 | 5.0 | 5.0 |

**Figure 5.1.:** Leakages reported by approxflow-java and ApproxFlow, rounded to one decimal place. The numbers in the column headers refer to the unrolling and inlining limits, the columns marked with $\infty$ show the results for sufficiently high limits, with black box handling disabled for approxflow-java (see section 5.2).
[1]Does not follow restrictions on black boxes (see above).
[2]ApproxFlow does not support public input.
[3]Did not terminate after 30 minutes or returned an error.

Figure 5.1 shows the leakages reported by approxflow-java and ApproxFlow. We briefly discuss these results. Both approxflow-java and ApproxFlow reported the correct leakage for *Parity*, even for insufficient unwinding limits. This is a trait of the test program, as it already leaks one bit even if only one loop iteration is executed. *Battlebits* uses public inputs, which ApproxFlow does not support. Approxflow-java reported the correct leakage. *Battlebits* demonstrates the stream-based input and

output model of approxflow-java, and its ability to deal with interleaved input and output.

For *Laundry* and *Recursive Laundry*, approxflow-java reported the correct leakage, as those examples actually leak the maximum possible amount through the black box. The leakages reported by ApproxFlow for *Laundry* with the unwinding limits 2, 8, and 32 were lower than the actual leakage, as the loop was not unwound sufficiently. To get the correct result through unwinding, the loop has to be unwound with 127 iterations. This is feasible with ApproxFlow, but as mentioned above, the input and output width were reduced to avoid the problem that we encountered with MaxCount. For a width of 32 or 64 bits, unrolling would not be feasible. For *Recursive Laundry*, ApproxFlow always returned a full leakage of 8 bits. The generated CNF file shows that the recursive function was not inlined recursively. The reported 8 bit leakage is therefore likely the result of a fallback behavior after aborting the inlining process.

The Partial Laundry example demonstrates a shortcoming of the black box approach: The leakage reported by approxflow-java was much higher than the actual leakage, as the black box approach assumes maximum flow through black boxes. Unwinding the loop with 16 iterations is sufficient to get the correct result, and this is reflected in the results reported by ApproxFlow. However, with unwinding, the burden of finding out that 16 iterations are sufficient falls to the user. The black box approach reports a sound, though overapproximated, result without requiring such judgment from the user. With black boxes disabled and at least 16 unroll iterations, approxflow-java returns the correct result.

The voting and smart grid test programs show similar behavior from ApproxFlow as the loop-based laundry test programs: For insufficient unrolling limits, the leakage is underapproximated, but if the limits are sufficiently high, the correct leakage is reported. For approxflow-java, an important difference to the behavior for *Partial Laundry* can be observed: while for *Partial Laundry*, black box handling has to be disabled to take advantage of a sufficiently high unrolling limit, this is not the case for the voting and smart grid programs. The reason for this difference is that the loops in the voting and smart grid programs abort after a fixed number of iterations. For approxflow-java this means that the black box is never reached. By contrast, the loop in *Partial Laundry* does not abort after 15 iterations; only the conditional statement in the body is not executed anymore. Approxflow-java can not know this as it does not analyze the loop body. For insufficient limits, approxflow-java reported a sound, but overapproximated leakage for the voting and smart grid programs, just as for the laundry programs.

**Runtime** Figure 5.2 lists the runtimes of approxflow-java and ApproxFlow for the tested configurations. The runtimes are the arithmetic means over five runs for each

configuration. The deviation was less than 10% across runs for all configurations. Approxflow-java had significantly higher runtimes than ApproxFlow for all test programs and configurations. The primary reason for this is that the CNF formulas generated by JBMC are much more complex than those generated by CBMC for equivalent programs. Both the number of clauses and the number of variables were at least ten times, typically more than 20 times as high.

Black box handling further increases the runtime, as the model checker and model counter are run multiple times. This also explains the lower runtimes for the runs without black box handling. Additionally, output streams increase the complexity of the analyzed program unnecessarily in cases where fixed output variables would suffice. Finally, maximum model counting, which we use not only for public input, but also for black box handling, is significantly slower than projected model counting. This amplifies the impact of black box handling. It also explains the high runtime of the *Battlebits* example, which is centered around public input.

| Program | approxflow-java | | | | ApproxFlow | | | |
|---|---|---|---|---|---|---|---|---|
| | 2 | 8 | 32 | $\infty$ | 2 | 8 | 32 | $\infty$ |
| Parity | 3.8 | 4.0 | 4.6 | 2.2 | 0.1 | 0.1 | 0.1 | 0.1 |
| Battlebits | -[1] | -[1] | -[1] | 66.2 | -[2] | -[2] | -[2] | -[2] |
| Laundry | 18.9 | 20.5 | 23.4 | -[3] | 0.1 | 0.2 | 1.9 | 4.1 |
| Recursive Laundry | 18.4 | 19.8 | 52.6 | -[3] | 0.1 | 0.1 | 0.1 | 0.1 |
| Partial Laundry | 19.1 | 21.0 | 23.5 | 1.9 | 0.1 | 0.1 | 0.4 | 0.1 |
| Voting 1 | 2.3 | 2.3 | 2.3 | 2.0 | 0.1 | 0.1 | 0.2 | 0.2 |
| Voting 2 | 2.5 | 2.7 | 3.2 | 2.4 | 0.1 | 0.2 | 0.8 | 0.8 |
| Smart Grid | 2.3 | 2.4 | 2.4 | 1.9 | 0.1 | 0.1 | 0.1 | 0.1 |

**Figure 5.2.:** Runtimes of approxflow-java and ApproxFlow in seconds, rounded to one decimal place. The numbers in the column headers refer to the unrolling and inlining limits, the columns marked with $\infty$ show the results for sufficiently high limits, with black box handling disabled for approxflow-java (see section 5.2).
[1]Does not follow restrictions on black boxes (see above).
[2]ApproxFlow does not support public input.
[3]Did not terminate after 30 minutes or returned an error.

# 6. Conclusion and Future Work

## 6.1. Conclusion

We presented a quantitative information flow analysis that uses bounded model checking and model counting but avoids some of the inherent limitations of that approach, specifically limited loop iterations and recursion depth. To our knowledge, it is the first analysis to do so. Furthermore, it supports input and output streams, whereas comparable analyses work on a simpler input and output model.

While our tool is not yet applicable to most real-world programs, it demonstrates a novel approach for handling programs with unbounded loops or recursion that may be used in future analyses. We believe that this, and the more general input and output model, are important steps towards the goal of using bounded model checking and model counting to analyze quantitative information flow in real-world programs.

## 6.2. Future Work

We see several opportunities for improving the presented analysis and tool regarding, runtime and supported programs.

While our analysis has several severe limitations with regards to the supported programs, many of those are not inherent to our approach. In particular, our general approach is not inherently limited to primitive inputs and outputs, nor does it inherently limit black boxes to local variables. Future analyses could possibly avoid those limitations while still using our black box approach.

The accuracy for programs with black boxes could possibly be improved through limited analysis of the code within the black box. In particular, analyzing the body of a loop or non-recursive parts of a recursive method may provide valuable information for the flow through the loop or method as a whole.

The runtime of approxflow-java depends primarily on the runtime of the model counter. As such, efforts to improve the runtime should in our opinion be focused on reducing the complexity of the CNF formula and the number of runs of the model counter. This could possibly be achieved by discarding irrelevant parts of the formula or the program, respectively. Furthermore, detecting the appropriate unrolling and inlining limits in simple cases, for example in numerical for loops, could avoid unnecessary black boxes.

# Bibliography


[1] 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, no. 6, pp. 280–287, 2014.

[2] G. Smith, "On the foundations of quantitative information flow," in *Foundations of Software Science and Computational Structures* (L. de Alfaro, ed.), (Berlin, Heidelberg), pp. 288–302, Springer Berlin Heidelberg, 2009.

[3] D. J. Fremont, M. N. Rabe, and S. A. Seshia, "Maximum model counting," Tech. Rep. UCB/EECS-2016-169, EECS Department, University of California, Berkeley, Nov 2016. This is the extended version of a paper to appear at AAAI 2017.

[4] B. Espinoza and G. Smith, "Min-entropy leakage of channels in cascade," in *Formal Aspects of Security and Trust* (G. Barthe, A. Datta, and S. Etalle, eds.), (Berlin, Heidelberg), pp. 70–84, Springer Berlin Heidelberg, 2012.

[5] 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.

[6] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (K. Jensen and A. Podelski, eds.), vol. 2988 of *Lecture Notes in Computer Science*, pp. 168–176, Springer, 2004.

[7] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. Pasareanu, "Symbolic quantitative information flow," *ACM SIGSOFT Software Engineering Notes*, vol. 37, pp. 1–5, 11 2012.

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

[9] 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.

[10] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa, "Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation," *Journal of Computer Security*, vol. 21, 02 2013.

[11] G. Smith, "Quantifying information flow using min-entropy," in *2011 Eighth International Conference on Quantitative Evaluation of SysTems*, pp. 159–167, 2011.

[12] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," 9 1992.

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

[14] R. A. Aziz, G. Chu, C. Muise, and P. Stuckey, "#∃SAT: Projected model counting," in *Theory and Applications of Satisfiability Testing – SAT 2015* (M. Heule and S. Weaver, eds.), (Cham), pp. 121–137, Springer International Publishing, 2015.

[15] M. Soos, S. Gocht, and K. S. Meel, "Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling," in *Computer Aided Verification* (S. K. Lahiri and C. Wang, eds.), (Cham), pp. 463–484, Springer International Publishing, 2020.

[16] M. Soos and K. S. Meel, "Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 1592–1599, Jul. 2019.

[17] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, "JBMC: A bounded model checking tool for verifying Java bytecode," in *Computer Aided Verification (CAV)*, vol. 10981 of *LNCS*, pp. 183–190, Springer, 2018.

[18] S. Falke, F. Merz, and C. Sinz, "The bounded model checker LLBMC," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 706–709, 2013.

[19] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT

calls," in *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 7 2016.

[20] F. Biondi, A. Legay, and J. Quilbeuf, "Comparative analysis of leakage tools on scalable case studies," 08 2015.

[21] M. Soos, "MaxCount." https://github.com/meelgroup/maxcount, 2019.

[22] J. Bechberger, "ApproxFlow." https://github.com/parttimenerd/approxflow, 2021.

[23] L. R. Ford and D. R. Fulkerson, *Maximal Flow Through a Network*, pp. 243–248. Boston, MA: Birkhäuser Boston, 1987.

# Erklärung

Hiermit erkläre ich, Yannick Urbach, dass ich die vorliegende Bachelorarbeit 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. Black Box Handling Using Flow Networks

This appendix discusses an alternative to the staged flow approach for handling black boxes described in section 3.2.3. As a starting point, we use the black box graph $G$ introduced in definition 8. We construct a flow network by introducing *edge leakage* as a capacity measure:

**Definition 11.** The *edge leakage* $l_e(b_i, b_j)$ is the *leakage* of the program resulting from the following modification of $p$:

- The black box sections $B$ are omitted.
- If $b_i \neq in$, private inputs of $p$ are replaced with *unknown* values, and the output values of $b_i$ are replaced with private input.
- The output values of all other black box sections are replaced with *unknown* values.
- If $b_j \neq out$, public outputs of $p$ are omitted, and the input values of $b_j$ are considered public output.

*Unknown* values are treated like public input, but do not affect the sequence output stream.

*Edge leakage* corresponds to the maximum amount of information that can flow along an edge of $G$. With the source $b_0 = in$ and the sink $b_{n+1} = out$, we get the flow network $N = (G, l_e, in, out)$. We can apply regular maximum flow algorithms such as the Ford-Fulkerson algorithm [23] to $N$ to get a *sound* approximation for the leakage of $p$. Figure A.1 demonstrates this approach for the example program introduced in section 3.2.3.

However, this example also shows a shortcoming of this approach: the bit that is passed directly from `in` to `out` is the same bit as the one passed to `b2`. Consequently,

```
int in  = privIn();

int r1  = b1(in & 0b11);
int r2  = b2((in & 0b01) | (r1 & 0b10));
int out = (in & 0b10) | (r1 & ~0b10);

pubOut(out);
```

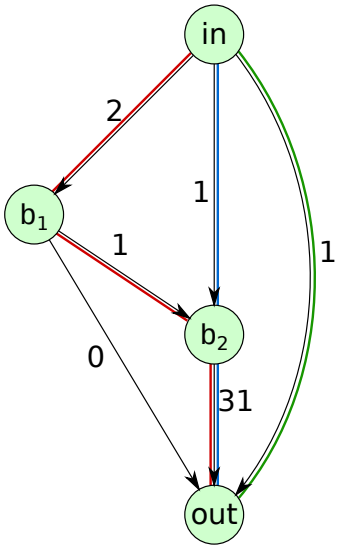



**Figure A.1.:** Example for using maximum flow to determine a sound approximation; augmenting flows are highlighted.

the total information leaked can be at most 2 bit, as the approach discussed in section 3.2.3 correctly determined. Approaches that only work on individual edges can not take this into account. This example therefore proves that there are cases where the staged flow approach provides a better approximation than the flow network approach.

Additionally, we will show that the approximation given by the staged flow approach is always at least as good as that of the flow network approach:

**Theorem 3.** Let maxflow $N$ be the maximum flow through $N$. Then

$$\min_{C_i} l_p(C_i) \leq \text{maxflow } N.$$

*Proof.* According to the max-flow min-cut theorem [23], maxflow $N$ is equal to the minimum capacity of all cuts. Due to the structure of $G$, the cut set of every cut other than the $C_i$ contains at least the same edges as some $C_i$, and therefore has at least the same capacity. Consequently, maxflow $N$ is equal to the minimum capacity of the $C_i$.

$l_p(C_i)$ is lower than or equal to the capacity of $C_i$, as all leakage in $l_p(C_i)$ must occur through one of the edges in the cut set of $C_i$.

Thus, $\min_{C_i} l_p(C_i) \leq \text{maxflow } N$. □

Note that the staged flow approach is also typically much faster than the flow network approach, as the number of modifications of $p$ that have to be analyzed is only linear in the number of black boxes, as opposed to quadratic for the flow network approach. On top of that, the flow network approach requires more *unknown* values, which also significantly impacts performance.

For these reasons, we consider the staged flow approach to be preferable.

# B. Evaluated Programs

This appendix contains the code listings for the programs used for Evaluation in chapter 5. For brevity, only the relevant methods are listed.

### Parity

```
static void main(String[] args) {
    long in = privIn();

    boolean parity = false;

    for (int j = 0; j < 64; ++j) {
        parity ^= (in & (1 << j)) != 0;
    }

    pubOut(parity);
}
```

### Battlebits

```
static void main(String[] args) {
    long board = privIn();

    for (int i = 0; i < 3; ++i) {
        byte shot = pubIn();
        assert shot > 0 && shot < 64;

        boolean hit = ((board >>> shot) & 1) != 0;

        pubOut(hit);
    }
}
```

### Laundry

```
static void main(String[] args) {
    int in = privIn();
    assert in > 0;

    int out = 0;

    for (int i = 0; i < in; ++i) {
        ++out;
    }

    pubOut(out);
}
```

### Recursive Laundry

```
static int launder(int in) {
    if (in <= 0) {
        return 0;
    }

    return launder(in - 1) + 1;
}

static void main(String[] args) {
    int in = privIn();
    assert in > 0;

    pubOut(launder(in));
}
```

### Partial Laundry

```
static void main(String[] args) {
    int in = privIn();
    assert in > 0;

    int out = 0;

    for (int i = 0; i < in; ++i) {
        if (i < 15) {
            ++out;
        }
    }

    pubOut(out);
}
```

### Voting 1

```
static void main(String[] args) {
    int voterCount = 31;

    byte result = 0;

    // bit vector of votes, truncated to match voter count
    long votes = privIn() & (-1 >>> (64 - voterCount));

    for (int i = 0; i < voterCount; ++i) {
        if (((votes >>> i) & 1) != 0) {
            ++result;
        }
    }

    pubOut(result);
}
```

### Voting 2

```
static void main(String[] args) {
    int candidateCount = 31;

    // results for candidates, two bits each
    long result = 0;

    // votes of the three voters
    byte voteA = privIn();
    byte voteB = privIn();
    byte voteC = privIn();

    for (int i = 0; i < candidateCount; ++i) {
        byte count = 0;

        if (voteA == i) {
            ++count;
        }

        if (voteB == i) {
            ++count;
        }

        if (voteC == i) {
            ++count;
        }

        result |= (count << (2 * i));
    }

    pubOut(result);
}
```

### Smart Grid

```
static void main(String[] args) {
    int totalCount = 16;
    int smallCount = 4;
    int mediumCount = 8;
    int largeCount = totalCount - smallCount - mediumCount;

    int smallConsumption = 1;
    int mediumConsumption = 2;
    int largeConsumption = 3;

    // bit vector of presence, truncated to match consumer count
    long present = privIn() & (-1 >>> (64 - totalCount));

    int globalConsumption = 0;

    for (int i = 0; i < totalCount; ++i) {
        if (((present >> i) & 1) != 0) {
            if (i < smallCount) {
                globalConsumption += smallConsumption;
            } else if (i < smallCount + mediumCount) {
                globalConsumption += mediumConsumption;
            } else {
                globalConsumption += largeConsumption;
            }
        }
    }

    pubOut(globalConsumption);
}
```