Upper Bound Computation of Information Leakages for Unbounded Recursion

Johannes Bechberger[1](✉) and AlexanderWeigl[2]

[1] Institute for Program Structures and Data Organization, Karlsruhe, Germany
[2] Institute of Information Security and Dependability (KASTEL), Karlsruhe Institute of Technology, Karlsruhe, Germany

Abstract. Confidentiality is an important security goal that is ensured by the absence of information flow between secrets and observable outputs. Quantitative information flow (QIF) analyses quantify the amount of knowledge an attacker can gain on the secrets by observing the outputs. This paper presents a novel approach for calculating an upper bound for the leakage of confidential information in a program regarding min-entropy. The approach uses a data flow analysis that represents dependencies between program variables as a bit dependency graph. The bit dependency graph is interpreted as a flow network and used to compute an upper bound for the leakage using a maximum flow computation. We introduce two novelties to improve the precision and soundness: We strengthen the precision of the data flow representation by using the path conditions. We add sound support of unbounded loops and recursion by using summary graphs, an extension of a common technique from compiler engineering. Our approach computes a valid upper bound of the leakage for all programs regardless of the number of loop iterations and recursion depth. We evaluate our tool against a state-of-the-art analysis on 13 example programs.

Keywords: Security analysis · Quantitative information flow · Bit dependency graphs

1 Introduction

Information Flow. The analysis of secure information flow (IF) tries to find the information flow of confidential secret information to output variables that can be observed by unclassified personnel or attackers. It is an important analysis to ensure the confidentiality of programs. Traditionally, the result of an IF analysis is a qualitative answer: either there is an influence of confidential information on attacker-observable outputs (we say the program leaks information) or not. Qualitative Information Flow is an established area of research that produced tools that scale to large programs and support a variety of language features.

```
input int h;
output int o := h % 2
```

Fig. 1. Program that leaks only one bit of information from the secret input h to the public output o.

The problem is that small leaks may often be acceptable and sometimes necessary; it is necessary to know the amount of leaked information. Secure Qualitative Information Flow cannot distinguish between a program that leaks only a single bit, like the program given in Fig. 1, and a program that leaks the whole secret, as the information flow is not quantified. The urge to distinguish between such cases leads to the need of quantifying the leakage.

Quantitative Information Flow (QIF) aims to calculate the leakage, the amount of secret information which is gained by an attacker, by executing a program. Applications range from ensuring the security of distributed applications to formally certifying data storage systems [19]. Typically, an attacker has access to the program code and can only see low outputs o after program termination. The quantified leakage of the program from Fig. 1 is clearly lower than the leakage of the program that leaks the whole secret. In the following, we call a QIF analysis *sound* if and only if the analysis computes an upper bound.

```
input int h;
int z := 0;
while (z ≠ h) {
    z := z + 1;
}
output int o := z
```

Fig. 2. *Laundering Attack* which leaks the secret input over all iterations of the loop.

Motivating Example. We consider now the program in Fig. 2 with signed fixed-width integers. This program demonstrates the *Laundering Attack*, and leaks the whole secret into the public output. This leakage occurs indirectly due to control statements. Each iteration of the loop itself only leaks the information whether $z = h$ for a specific z, but all iterations together leak the whole secret. Figure 2 is an example of how a small leak can be extended into a leak of the secret. A related real-world example is the brute-force attack on passwords, checking all possible passwords to find the correct one, with each call to the password check routine leaking only a small amount of information.

Many static QIF analyses based on abstract interpretation, model counting, or program algebras were proposed in recent years. They have in common that they only investigate programs up to a prior set upper bound on execution paths. If the upper bound is too small, the estimated leakage might be too low. This can be observed in our evaluation in Sect. 7. The usage of a prior set upper bound means that current analyses can only consider a limited number of loop iterations. There are multiple static analyses that support both loops and functions, but in practice, only a limited recursion depth and a limited number of loop iterations can be soundly analyzed due to resource limitations. As a result, these analyses cannot give, in practice, an upper bound for the leakage of all analyzed programs, an example for such a program is the Laundering Attack in Fig. 2.

Contribution. We present *Nildumu*[1], a novel over-approximative static QIF analysis for a while-language with recursion and fixed-length arrays with copy semantics. It supports both unbounded loops and recursion, contrary to the current state of the art. The analysis does this by considering a limited number of execution paths, like previous approaches, over-approximating the effects of the remaining iterations and recursive calls. The basis of the analysis is a bit dependency graph, which records the dependencies between the values in a program on the bit-level. During the construction of the graph, path conditions are taken into account. The bit dependency graph is extended using the novel summary graphs, which are used to improve the precision of the over-approximation. The evaluation shows that the analysis is approximately as precise as current analyses based on model counting while being sound for every number of considered execution paths.

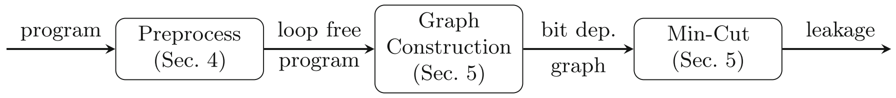


Fig. 3. Structure of our analysis.

Overview. We first describe the related static analyses in Sect. 2 and the theoretical foundations in Sect. 3. We then describe our analysis itself. The analysis is structured into different parts, as seen in Fig. 3: The program is first transformed into an equivalent loop-free program with recursion, lowering arrays to int variables. This loop-free program is then transformed into a simplified form so that variables are only assigned once. These transformations are given in Sect. 4. Then the summary graphs are computed for every function, and with them, the bit dependency graph is created, presented in section Sect. 5. The actual computation of the leakage is then based on the bit dependency graph, using a maximum flow computation. We then give an improvement of the precision of the analysis harnessing the knowledge gathered from path conditions in Sect. 6. We follow this by an evaluation in Sect. 7 comparing the analysis with a state-of-the-art model counting analysis and end with the conclusion and future work in Sect. 8.

2 Related Work

There are multiple static QIF analyses based on abstract interpretation, as presented by Smith in [27], like *jpf-qif* developed by Phan et al. [26]. Recent advances in the field of approximative model counting resulted in the development of analyses that can process code written in C and C++, like *ApproxFlow* [4]. In contrast to the SAT-based model counting, *Moped-QLeak* [9] uses binary decision

[1] Nildumu is Lojban for "is a quantity".

diagrams (BDD) for computing a summary of a program and using this summary to compute its leakage. Model-counting and BDD-based analyses rely on inlining and unwinding and are prone to under-approximations, as mentioned in the introduction.

Furthermore, there are two static analyses by Mu [23] and Clark et al. [12] that use a Program Dependence Graph (PDG) to track the dependencies between variables in a graph representation of the program. This differs from our approach, which tracks the dependencies between individual bits instead of variables. These analyses compute the value probability distributions for each program variable. Both analyses are based on the notions given in [18], describing an algebra for an imperative language. The analyses using these techniques can soundly analyze programs of a while-language using a probabilistic denotational semantic. But these analyses do not support recursion and are limited to small programs. Newer approaches [1] improve on these analyses based on newer work on the formalization of hyperproperties, but they are not yet implemented in tools. The advantage of these approaches is that they support multiple leakage measures.

Finally, although there is no other bit dependency graph based analysis, there is one dynamic analysis using byte dependency graphs: The dynamic analysis by McCamant and Ernst [21] uses dynamic tainting instead of statically tracking the flow of information through the program with a byte level granularity. Other dynamic analyses exist, but they are usually based on black box approaches [10,11] that do not consider the program code at all.

As stated previously, none of these tools support the static analysis of programs with both arbitrary numbers of loop iterations and recursion depth.

3 Foundations

We use the information-theoretical notion of QIF as presented by Smith [28]. The entropy $H(X)$ describes the amount of information of a random variable X. It gives the minimal number of bits that are required to encode the information of X (Shannon Entropy).

In the following, we consider only sequential programs, similar to [28], where the attacker only observes the output `O` after the execution of the program finishes and has no information on the secret input `H`. In QIF, we are interested in the information shared between `H` and `O`. This information is called *mutual information*. It is denoted as $H(\mathtt{H};\mathtt{O})$ and expresses the information gained on `H` by observing `O`. The actual leakage $I(\mathtt{H};\mathtt{O})$ is then defined as the reduction of the uncertainty by observing `O`: $I(\mathtt{H};\mathtt{O}) = H(\mathtt{H}) - H(\mathtt{H};\mathtt{O})$.

In the following, we use the min-entropy H_∞, which is based on the concept of vulnerability [28] and quantifies the probability that the secret is guessed by the attacker in one try. Formally, the vulnerability $V(\mathtt{H})$ is defined as $V(\mathtt{H}) = \max_{h \in \mathcal{H}} P[\mathtt{H} = h]$ with the resulting entropy being $H_\infty(\mathtt{H}) = \log \frac{1}{V(\mathtt{H})}$ [17]. In particular, $V(\mathtt{X})$ is the worst-case probability that `X`'s value can be guessed correctly in one try.

Assuming we have a deterministic program with a uniformly distributed secret, the min-entropy leakage I_∞ is calculated by counting the different possible outputs of a program [28]. Formally, let O be the set of possible outputs of the program, then the leakage is

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

This leakage is an upper bound of the leakage over all distributions of H.

Soundness. To work with estimations of static QIF analyses that are not exact, we define *soundness* as follows: An analysis is *sound* if and only if the calculated leakage for all programs p with the secret input H and public output O, $\hat{I}^p(\mathtt{H};\mathtt{O})$, is an upper bound of the actual leakage $I^p(\mathtt{H};\mathtt{O})$, i.e., $I^p(\mathtt{H};\mathtt{O}) \leq \hat{I}^p(\mathtt{H};\mathtt{O})$.

Program Dependency Graph (PDG). A PDG is a data dependence graph with added control flow edges [15]. Such a graph consists of nodes that represent variables and operations. There is an edge from a node a to a node b present in this graph if the value of b directly depends on the value of a (data dependence) or if the value of a directly affects whether or not b is executed (control dependence). Our analysis uses a PDG as its underlying representation of the program structure.

Constant Bit Analysis. For our QIF analysis, we exploit a static intra-procedural constant bit analysis on a PDG. A constant bit analysis aims to find bits that are statically known. We base our analysis on the analysis described by Budiu et al. [7] which uses a *bit lattice* ($\mathbb{B}$). This lattice contains the possible statically known information on a bit. A bit is a constant (0 or 1), might be both ($\top_\mathbb{B}$), or is never evaluated ($\bot_\mathbb{B}$).

A constant bit analysis associates each node in the PDG with a tuple of elements from the bit lattice representing the knowledge that we have of each bit of the value of each node.

4 Preprocessing

Shape of Programs. In this paper, we consider programs of a while-language containing the typical imperative statements: assignments, if-statements and while-loops (cf. Fig. 4). Moreover, the programming language contains functions that might be directly or mutually recursive. Also, functions can have multiple return values, an assignment of the form $(v_1, \ldots, v_k) \ := \ f()$ allows to assign the return values of the function call $f()$ to multiple variables v_1 to v_n. Additionally, the dot denotes bit-access operator, i.e., $e.n$ denotes the nth bit of the expression e. To identify the secret and public information, variables declaration can contain the modifier *input* (secret) and *output* (public). All variables without such a modifier are considered as hidden and non-confidential.

The only supported data types are signed fixed-sized integers and fixed-length arrays. Integers are represented in two's complement with an arbitrary but fixed bit-width called W in the following. Boolean values are represented by the integers 0 and 1.

Preprocessing. We start the analysis by preprocessing programs into an array- and loop-free form to simplify the QIF analysis. Arrays have a fixed-length, and therefore can be split into single variables that represent its entries. This technique is known as *scalar replacement of aggregates* [24]. Every access of an array element with a constant index can directly be mapped onto the corresponding variable. All other accesses are replaced with if-else-cascades to determine the correct variable.

Loops are transformed into recursive functions. The transformation rule is given in Fig. 5 which requires multiple return values. The application of this rule for the example in Fig. 2 is in Fig. 6. This transformation is followed by the inlining of all functions on their call-sites with argument passing and return statements replaced by variable assignments. Recursive functions are only inlined up to a user-specified bound. This is a common technique that is used in model checking and program analyses to support functions [3,24]. Note that the function calls are preserved when the inlining bound is hit. Thus, the resulting program is not free of function calls, which are handled later in our QIF analysis by over-approximating the behavior of the remaining (recursive) function calls. The inlining increases the precision of the analysis as every inlined function call is not over-approximated and increases its run-time.

After the inlining, we translate the program into Static Single Assignment form (SSA). We introduce fresh variables, such that every variable is only assigned once. Moreover, we ensure that the right-hand side of each assignment is an atomic expression. An atomic expression is either a function call or a binary operator with variables (v) or constants (n) as operands. The final result is a program that only consists of if- and function call statements, as well as assignments $v = e$ and return statements `return` e where e is an atomic expression.

5 Bit Dependency Graph

This section covers the novel generation of the bit dependency graph for a program with arbitrary recursive functions (Sect. 5.1, Sect. 5.3) but without loops and arrays. The construction is based on the constant bit analysis and results in

$$
\begin{array}{lll}
\mathit{Var.\ def} & V & ::= (\mathbf{input} \mid \mathbf{output})^{?}\ t\ v \\
\mathit{Func.\ call} & C & ::= f(E_1, \ldots, E_m) \\
\mathit{Expression} & E & ::= v \mid n \mid E \odot E \mid E.n \\
\mathit{Statements} & S & ::= V \mid v := E \mid (v_1, \ldots, v_k) := C \mid v[E] := E \mid \\
 & & \quad\ \ S;\ S \mid \mathbf{while}\ (E)\ \{\ S\ \} \mid \mathbf{if}\ (E)\ \{\ S\ \}\ (\mathbf{else}\ \{\ S\ \})^{?} \\
\mathit{Func.\ def} & F & ::= t_1\ \ldots\ t_k\ f(t_{p1}\ v_1,\ \ldots,\ t_{pm}\ v_m)\ \{\ S;\ \mathbf{return}\ E_1, \ldots, E_k\ \} \\
\mathit{Program} & P & ::= F^{*}\ S
\end{array}
$$

Fig. 4. The grammar of the considered while-language. Placeholder v denotes a variable name, t a type name, n an integer constant and $\odot$ a typical binary operator like addition, multiplication, or exclusive-or on integers.

```
                 $t_{w_1} \dots t_{w_n}$ $f_l$ ($t_{w_1}$ $w_1$, ..., $t_{w_n}$ $w_n$, $t_{r_1}$ $r_1$, ..., $t_{r_m}$ $r_m$) {
                   if (E) {
while (E) {   ⇒      S; $(w_1, \dots, w_n)$ := $f_l(w_1, \dots, w_n, r_1, \dots, r_m)$ }
  S                return $w_1, \dots, w_n$
}                }
                 $(w_1, \dots, w_n)$ := $f_l(w_1, \dots, w_n, r_1, \dots, r_m)$
```

Fig. 5. Translation of a loop into a semantically equivalent function, with $w_1, \dots, w_n$ being the variables written in the loop and $r_1, \dots, r_m$ being other variables that are accessed in the loop.

```
                        int $f_l$(int $z$, int $h$) {
                          if ($z \neq h$) {
while ($z \neq h$) {  ⇒       $z$ := $z + 1$; $z$ := $f_l$ ($z$, $h$) }
  $z$ := $z + 1$              return $z$
}                       }
                        $z$ := $f_l$($z$, $h$)
```

Fig. 6. Translation of the loop from Fig. 2 using the schema from Fig. 5.

a graph that expresses the dependencies between single bits. We use this graph to approximate the leakage of a program (Sect. 5.2).

Definition 1 (Bit Dependency Graph). *A bit dependency graph $G = (V, E)$ is a directed graph of nodes where each node represents a bit that belongs to the value of a node in the underlying PDG. This graph contains an edge from v_1 to v_2 if v_2 data or control depends on v_1.*

Each bit node in the graph represents a single bit of the information on a variable at a specific program location, due to the SSA form. Thus, a bit dependency graph represents the dependencies between variables at the bit-level. A peculiarity in this definition is that an edge between two nodes (bits) expresses the possibility of a dependency. To achieve a sound analysis, the set of distinct paths between nodes must always be a superset of the actual bit dependencies defined by the program.

In the following, we call nodes that are reachable from a node $v \in V$ the *transitive successors* of v and nodes $v_i \in V$ for which an edge $(v, v_i) \in E$ exists *successors* of v.

Construction. We construct a bit dependency graph from a PDG during the constant bit analysis by collecting the dependencies between the individual bits associated with the PDG nodes.

Let $G = (V, E)$ be the bit dependency graph for a given preprocessed program. The preprocessed program only consists of control statements, function calls, atomic assignments and return statements, as described before. The set of vertices of the graph is $V := \{x.i \mid 1 \leq i \leq W,\ x \text{ is a PDG node}\}$ that contains a node for each bit of every PDG node and thereby every variable in the program. The set of edges E is formed by using specific function handlers. A function handler, defined in Sect. 5.1, models a bit dependency graph from the arguments to the return value. We treat every operator in the following as an implicitly defined function, e.g. $a + b$ is treated as $f_+(a, b)$. Let us consider the case of two arguments in function calls (or binary operators), $x := f(y, z)$ (or $x := y \odot z$): We add edges between the bit nodes of y, z and x if there is a data or control dependency between them. We can trivially extend this to functions with higher arity.

Due to the preprocessing, we only need to consider assignments with function calls and single operator expressions. Each specific function or operator requires a function handler.

Example 1. The bit dependency graph for the program x := y | z with two bit integers is given in Fig. 7. The nodes y_i and z_i are connected to x_i since each bit of the result depends on the corresponding bits of the operands.

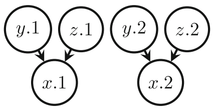


Fig. 7. Graph for x := y | z

In the remaining, we define the concept of function handlers (Definition 2) and discuss different handlers (Sect. 5.1 and Sect. 5.3). We afterwards state the relationship between the bit dependency graph and the leakage computation Sect. 5.2.

5.1 Handling Functions

We formally introduce the concept of a function handler h_c, which models the bit dependencies from the arguments $a_1, \ldots, a_n$ to the return value x. A function handler returns a specific bit dependency graph for a specific function call $x := f(a_1, \ldots, a_n)$. Therefore, the handler can react to specific arguments, e.g. an optimization for neutral elements of operators are possible $(a + 0 = a)$.

In the best case, this function handler represents the bit dependencies precisely. In the worst case, if no such function handler exists, we add an approximative sub-graph. Such an approximative sub-graph leads to a sound analysis if it is an over-approximation, i.e., it adds at least as many distinct paths between every parameter node and every return node as the precise sub-graph.

Definition 2 (Function Handler). *A function handler for a specific function call c, $x := f(a_1, \ldots, a_n)$, with the tuple of bit nodes $A_c = (a_1.1, \ldots, a_1.W, \ldots, a_n.1, \ldots, a_n.W)$ related to the arguments $a_1, \ldots, a_n$, is formally defined as a function $h_c : A_c \mapsto (V_c, E_c)$ with $A_c \subseteq V_c$.*

The resulting graph $G_c = (V_c, E_c)$ is an over-approximation of the application of f and the return value nodes R_c are used as the nodes of x.

We distinguish two kinds of handlers: the *built-in* and the *summary* handlers. We describe in the following the built-in handlers and detail the *summary* handlers in Sect. 5.3.

Built-In Handler. For operators and built-in functions, we define handlers that model their effect: We interpret non-bitwise operators as their equivalent combination of bit-wise operators and over-approximate more complex operators like multiplication. This allows the analysis to only implement the bit-wise operators directly. The built-in handlers are more precise than the summary handlers but have to be implemented directly in the core analysis.

5.2 From Bit Dependency Graph to Leakage

We can compute an approximation of the leakage by using network flow algorithms. Commonly, a directed node-weighted flow network $N_G = (G = (V, E), \gamma, v_{source} \in V, v_{sink} \in V)$ consists of a directed graph G, a node capacity $\gamma \colon V \to \{1, \infty\}$ and a source and a sink node for the flow. A comparable idea based on an edge-weighted flow network has first been used by McCamant and Ernst [21].

Construction of the Flow Network. Given a bit dependency graph $G = (V, E)$ with the nodes $V_{input} \subseteq V$ representing the secret input bits and the nodes $V_{output} \subseteq V$ representing the public output bits, we can construct the corresponding node-weighted flow network N_G as follows: We introduce a new source node v_{source} which has as successors all input nodes V_{input} and a new sink node v_{sink} which is a successor of all output nodes V_{output}:

$$N_G = (G' = (V', E'), \gamma, v_{source}, v_{sink}) \qquad V' = V \cup \{v_{source}, v_{sink}\}$$
$$E' = E \cup \{\, (v_{source}, v) \mid v \in V_{input} \,\} \cup \{\, (v, v_{sink}) \mid v \in V_{output} \,\}$$
$$\gamma(v) \mapsto \begin{cases} \infty & : v \in \{v_{source}, v_{sink}\} \\ 1 & : otherwise \end{cases}$$

Theorem 1 (Leakage Computation using Minimum Cuts). *The size of the minimum node cut of the network N_G is an upper bound of the leakage of a program with the bit dependency graph G.*

Proof Sketch. First two observations: A single bit can only be statically unknown if it is either a secret input bit or it transitively depends on at least one secret input bit. Consider now the bits $b_1, \ldots, b_n$ that form the bit vector b which are statically unknown. b can than have at most 2^n different values at runtime.

If we can find the bits $b'_1, \ldots, b'_m$ so that all paths from v_{source} to $b_1, \ldots, b_n$ contain these bits, then b can have at most 2^m values: The vector b' can have at most 2^m values and every value of b' leads to one value of b at runtime.

The minimum cut M is the b' with the minimal combined weight if we consider b to be the vector of public output bits. $2^{|M|}$ is therefore an upper bound on the number of different output values at runtime and as a result M is an upper bound for the min-entropy of the underlying program (see Sect. 3).

Computation. We can compute the minimum node cut by transforming the node-weighted network into an edge-weighted network [14, Algorithm 9] on which we compute the minimum edge cut. The minimum edge cut can be computed by using maximum flow algorithms as a result of the max-flow min-cut theorem [6]. Another possibility is to use a Partial MaxSAT solver as presented in Sect. 6.

5.3 Summary Function Handler

By treating bit dependency graphs as node-weighted network flow graphs, we can reconsider function handlers and define *summary* handlers. We first define the concept of summary graphs, their construction, and at last their application in form of a function handler.

Definition 3 (Summary Graphs). *A summary graph G_s for a bit dependency graph G of a function f consists of the parameter nodes P, the return nodes R and the intermediary nodes Γ. The edges of G_s and Γ, satisfy the following constraint: The information flow between P and R is the same in G_s as in G.*

Summary graphs are modeled after the transitive dependence graphs for functions introduced by Horwitz et al. [16]. These dependence graphs consist of summary edges and are commonly used in compiler engineering for program slicing. A summary edge connects a parameter node with a return node if and only if there is a transitive dependency between them. A summary graph is a transitive dependence graph on bit-level that includes the nodes from the minimum-node-cut as intermediary nodes Γ to improve the precision.

Construction. Minimal summary graphs for each function are constructed iteratively using a fixed-point iteration over the call-graph. It uses a graph without any edges as a starting point for every function. The fixed-point iteration computes the summary graph for a given function f in each iteration using the following steps:

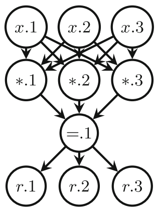
1. Construct the bit dependency graph G for f with parameters as secret inputs and return values as public outputs, using the current iteration's summary graphs whenever a function is called.
2. Reduce the graph G with parameter nodes P and return nodes R: Construct the flow network N_G and compute the minimum node cut Γ. Reduce the graph to a graph $G' = (V', E')$ that consists of $V' = P \cup R \cup \Gamma$ and transitive edges between P and $R \cup \Gamma$, and Γ and R.
3. Set $G = G'$ for the next iteration of the summary graph for f.

Using this construction, the summary graphs for all functions in the program can be pre-computed. Every iteration of the fixed-point iteration in the construction adds at least one new distinct path between the parameter and return nodes of at least one function. The fixed-point iteration terminates, as the number of distinct paths is bounded. Therefore, the construction itself terminates.

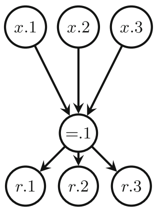
Example 2. We consider the function f given in Fig. 8a with three bit integers. Figure 8b shows the graph $\mathcal{G}(f)$ for the function and the resulting summary graph in Fig. 8c. This shows how the size of the summary graph is reduced.

```
int f(int x){
    int r := 0;
    if (x * 3 = 1){
        r := 0b111
    }
    return r
}
```

(a) Example program



(b) Function graph



(c) Summary graph

Fig. 8. Example function with its graphs, omitting constant nodes.

Summary Handler. The summary handler is a function handler which uses a copy of the precomputed summary graph G_s^f for a function f on its call-site c.

$$h_c^{summary} : A_c \mapsto \text{copy}(G_s^f)[P \mapsto A_c]$$

Summary function handlers summarize the effect of a function on the leakage computation. They are still an over-approximation as they cannot use the information the constant bit propagation has on the arguments at any given call-site.

Soundness. We follow with Menger's Theorem for directed graphs [6] that the minimum node cut is equivalent to the number of internally node disjoint paths, as all nodes have weight 1. By the construction of the reduced graph G', the disjointedness of paths is preserved by the graph reduction, thus the set of disjoint paths is a superset of the disjoint paths of G. The summary handler is therefore sound.

6 Increasing the Precision

```
if (x & 1 = (y >> 1) & 1){
    z := x
}
```

Fig. 9. Example for path conditions

Knowledge from path conditions is not used in the construction of the bit dependency graph as described in Sect. 5. We extend the previous graph construction to take this knowledge into account, which increases the precision of our analysis. We annotate each bit node b with a function $repl_b : \mathbb{B} \rightarrow 2^{2^{Bit}}$ which returns the sets of bits that can be considered equal under the assumption that b has a given value. We use these functions to compute the equal bits for every path condition. In particular $repl_{cond}(1)$ returns the bits that are considered equal in the current context under the assumption that *cond* evaluates to true.

For example, we know that `x & 1 = (y >> 1) & 1` evaluated to `true` in the then branch of the if-statement in Fig. 9, therefore we can infer that the first bit of `x` is equivalent to the second bit of `y`, $\{x.1, y.2\} \in repl_{\texttt{x\&1=(y>>1)\&1}}(1)$. The propagation of knowledge is based on the notion of propagated predicates, first formalized by Wegbreit [29]. Every path condition leads to new knowledge on bits.

This knowledge leads us to a set of bit dependency graphs, as we know in each context-specific bits that can be replaced by other bits, e.g. the bits belonging to `x.1` with the bits belonging to `y.2` in the example above. Inserting an edge from either bit in the specific context leads to a sound over-approximation. Therefore every of the possible graphs leads to an over-approximated leakage. In our example, we can replace the edge $(v_{x.1}, v_{z.1})$ with the edge $(v_{y.2}, v_{z.1})$ leading us to a set of graphs. We can either use simple heuristics to choose a specific graph or use a Partial MaxSAT (PMSAT) solver for leakage computation to optimize the chosen edges (Sect. 6) to minimize the calculated leakage.

Heuristic-Based Graph Selection. We select the graph, which promises the smallest leakage, by applying a simple greedy edge selection heuristic: In principle, we prefer edges that start in constant bits. The idea is that it improves the constant propagation and the precision of the analysis, as constant bits do not depend on the secret input. The advantage of this heuristic is its computational simplicity. Its main disadvantage is that it does depend on one of the possible edges starting in a constant bit, arbitrarily choosing an edge otherwise, not guaranteeing an optimal result. A preliminary evaluation showed that this did not affect the precision of the analysis for the programs in the evaluation. This heuristic is therefore used in the evaluation.

PMSAT-Based Leakage Computation. In general, an instance of PMSAT consists of formulas in conjunctive normal form (CNF) that consist of soft and hard clauses conjunctively combined with disjunctively connected (negated) propositional variables. A PMSAT solver, like Open-WBO [20], tries to find a satisfying variable assignment such that the variable assignment meets all given hard clauses, and the most possible soft clauses [8]. Finding such a solution is NP-complete but its usage removes the need for heuristics for incorporating the knowledge on replacement edges.

In the following, we give the encoding of the node-weighted flow network $N = (G' = (V', E'), \gamma, v_{source}, v_{sink})$ into hard and soft constraints: For each vertex v, we introduce the propositional variables c_v and r_v, which represents the participation of the vertex in the minimum cut: If c_v holds, add vertex v to the minimum cut, or if r_v holds, cut the graph after the successors v_s of vertex v (c_{v_s}) or their successor transitively. The hard constraints $\varGamma(v)$ for every node v are therefore defined as:

$$\varGamma(v) := \quad \underbrace{d_v \rightarrow (c_v \vee r_v)}_{(1)} \quad \wedge \quad \underbrace{r_v \rightarrow \bigwedge_{s \in \text{successors}(v)} (d_s \vee \bigvee_{i=1}^{n} d_{s_i})}_{(2)}$$

We create the helper variable d_v in (1) that states that we cut the graph either at the vertex or after the vertex. (2) states that if we consider cutting the graph after v then we have to either cut the graph after every successor s. If there are any replacements (v, s_i) for the edge (v, s), we can cut at or after any of the s_i instead. If v does not have any successors, $\Gamma(v)$ degenerates to $d_v \rightarrow c_v$.

We add the hard constraints $\neg c_{v_{source}}$ and $\neg c_{v_{sink}}$ as the source and the sink cannot, by definition, be part of the minimum cut. We add $r_{v_{source}}$ as the minimum cut consists of transitive successors of v_{source} and $\neg r_{v_{sink}}$ as we cannot cut after the sink. We finally add the soft constraint $\neg c_v$ for every node v, leading us to the final formula:

$$\underbrace{\bigwedge_{v \in V'} \Gamma(v) \wedge \neg c_{v_{source}} \wedge \neg c_{v_{sink}} \wedge r_{v_{source}} \wedge \neg r_{v_{sink}}}_{\text{hard}} \quad \wedge \quad \underbrace{\bigwedge_{v \in V'} \neg c_v}_{\text{soft}}$$

A PMSAT solver tries to maximize the number of fulfillable $\neg c_v$ clauses and thereby minimize the number of nodes participating in the minimum cut, leading us to a leakage computation.

7 Evaluation

We compare *Nildumu*[2] with *ApproxFlow*[3]. As stated before, we found no other tool that supports both unbounded loops and unbounded recursion and use ApproxFlow as a state-of-the-art analysis. ApproxFlow is based on model counting. It works by first creating a SAT formula representing a program using CBMC [13] and then counting the number of different assignments for the output variables using an approximate model counter. Although we do compare the runtimes of both tools, the value of the comparison is limited, as both tools are based on different libraries using different language runtimes.

Tool Configuration. The tools are evaluated with different levels of inlining and unwinding to show the effect of this parameter on the approximated leakage. We consider 2, 8, and 32 as both unrolling and inlining levels. A level of 32 is the default for ApproxFlow. ApproxFlow is by default configured so that its results differ by at most 0.8 bits from the real leakage with a probability of 80%, as ApproxFlow uses an approximate model counter. We use the same inlining levels in combination with the summary handler for Nildumu. For the sake of completeness, we also present the datapoints for Nildumu without path conditions support (32^w).

[2] Nildumu is available as open-source with a GUI at https://github.com/parttimenerd/nildumu with the full evaluation reproducible using the docker image `parttimenerd/nildumu`. The evaluation used version `49ebe88948874`.

[3] We used a modified version of ApproxFlow [4] with an update to ApproxMC4, publicly available at https://github.com/parttimenerd/approxflow.

Table 1. The computed leakage for all benchmarked programs with different unrolling levels k. The timeout was 2 h, timeouts are marked with a dash ("**-**"). Under-approximations of programs are marked as bold and underlined and over-approximations larger than one bit are marked as overlined, as a deviation of 0.8 bits is accepted for ApproxFlow with the default configuration. The second column I gives the actual leakages of the programs with $\sim$ marking the estimate by ApproxFlow as explained before and the third column I_{max} gives the maximum possible leakage, considering only the number of input and output bits.

Program	I [bit]	I_{max} [bit]	ApproxFlow			Nildumu			
			k=2	8	32	2	8	32	32^w
Laundering Attack	32.0	32.0	$\underline{\mathbf{1.6}}$	$\underline{\mathbf{3.2}}$	$\underline{\mathbf{5.0}}$	32.0	32.0	32.0	32.0
Binary Search (N=16)	16.0	32.0	$\underline{\mathbf{2.0}}$	$\underline{\mathbf{8.0}}$	16.0	$\overline{32.0}$	$\overline{32.0}$	16.0	$\overline{32.0}$
Binary Search (N=32)	32.0	32.0	$\underline{\mathbf{2.0}}$	$\underline{\mathbf{8.0}}$	32.0	32.0	32.0	32.0	32.0
Electronic Purse	2.0	32.0	$\underline{\mathbf{1.6}}$	2.0	2.0	$\overline{5.0}$	$\overline{5.0}$	$\overline{5.0}$	$\overline{32.0}$
Illustrative Example	4.1	32.0	4.1	4.1	4.1	5.0	5.0	5.0	$\overline{24.0}$
Implicit Flow	2.8	32.0	2.8	2.8	2.8	3.0	3.0	3.0	3.0
Masked Copy	16.0	32.0	16.0	16.0	16.0	16.0	16.0	16.0	16.0
Mix and Duplicate	16.0	32.0	16.0	16.0	16.0	16.0	16.0	16.0	16.0
Population Count	5.0	32.0	5.0	5.0	5.0	$\overline{10.0}$	$\overline{10.0}$	$\overline{10.0}$	$\overline{10.0}$
Sanity Check	4.0	32.0	$\overline{31.0}$	$\overline{31.0}$	$\overline{31.0}$	$\overline{32.0}$	$\overline{32.0}$	$\overline{32.0}$	$\overline{32.0}$
Sum	32.0	32.0	32.0	32.0	32.0	32.0	32.0	32.0	32.0
Smart Grid	$\sim$ 7.0	32.0	$\underline{\mathbf{1.6}}$	$\underline{\mathbf{3.1}}$	$\underline{\mathbf{5.5}}$	$\overline{32.0}$	$\overline{32.0}$	$\overline{32.0}$	$\overline{32.0}$
Preference Ranking (N=5)	$\sim$ 132.2	160.0	$\overline{160.0}$	132.1	132.1	$\overline{160.0}$	$\overline{135.0}$	$\overline{135.0}$	$\overline{135.0}$
Preference Ranking (N=10)	$\sim$ 294.4	320.0	$\overline{320.0}$	$\overline{320.0}$	**-**	$\overline{320.0}$	$\overline{320.0}$	$\overline{301.0}$	$\overline{301.0}$
Single Preference (N=5)	$\sim$ 8.0	160.0	$\overline{98.6}$	8.0	8.0	$\overline{160.0}$	$\overline{25.0}$	$\overline{25.0}$	$\overline{25.0}$
Single Preference (N=10)	$\sim$ 7.9	320.0	$\overline{258.6}$	$\overline{77.7}$	$\overline{17.7}$	$\overline{320}$	$\overline{320.0}$	$\overline{100.0}$	$\overline{100.0}$

Table 2. The mean execution time for all benchmarked programs in seconds. The timeout was 2 h and the standard deviation was at maximum 10%.

Program	ApproxFlow			Nildumu			
	k=2	8	32	2	8	32	32^w
Laundering Attack	0.1	0.1	0.2	1.4	1.4	2.0	1.4
Binary Search (N=16)	0.1	0.2	0.3	3.6	3.6	6.5	3.4
Binary Search (N=32)	0.1	0.2	0.6	3.6	3.6	7.2	3.5
Electronic Purse	0.2	0.3	3.8	1.3	1.5	1.2	2.0
Illustrative Example	0.1	0.1	0.1	0.6	0.6	0.7	0.6
Implicit Flow	0.1	0.1	0.1	0.7	0.7	0.9	0.7
Masked Copy	0.2	0.2	0.2	0.5	0.5	0.8	0.4
Mix and Duplicate	0.2	0.2	0.2	0.5	0.5	0.8	0.5
Population Count	0.1	0.1	0.1	0.7	0.6	0.9	0.6
Sanity Check	0.3	0.3	0.3	0.6	0.6	0.9	0.6
Sum	0.3	0.3	0.3	0.6	0.5	0.7	0.5
Smart Grid	0.2	0.3	0.3	53.1	101.9	248.9	36.6
Preference Ranking (N=5)	11.8	154.	422.7	144.4	149.4	148.4	274.3
Preference Ranking (N=10)	377.6	4061.7	**-**	402.7	442.0	819.8	2442.9
Single Preference (N=5)	6.0	1.2	1.2	43.7	43.8	44.1	293.0
Single Preference (N=10)	16.6	197.8	3520.7	162.5	163.9	163.8	3373.4

Benchmark Process. Both tools are run 5 times for every combination of program and unrolling level to account for randomness in the underlying system and in ApproxFlow. The benchmarking took place on an Intel Xeon Gold 6230 CPU with 512 GiB of RAM, running a Linux 5.4.0 kernel with OpenJDK 1.8.0 and CBMC 5.21.0. Both tools are restricted to two cores.

Benchmark Programs. We use the Laundering Attack from Fig. 2 and the commonly used benchmarks described in [2,9,22,25]. These benchmarks from literature can be categorized into programs that are focused on the handling of loops (Binary Search [22] and Electronic Purse [9]), the handling of conditions (Illustrative Example [22] and Implicit Flow [22]), the handling of bit operations (Mix and Duplicate [25] and Population Count [25]), and the handling of arithmetic or comparison operations (Sanity Check [25] and Sum [2]). We omit programs that use features not supported by the compared analyses.

We additionally use the Smart Grid and E-Voting examples from [5]. We use two versions of the E-Voting example as used by [4]: Ranking and single preference-based voting. There are no exact leakages known for these larger programs a priori, as the leakage depends on multiple configuration parameters. To estimate the exact leakage for the Smart Grid and E-Voting examples, we used ApproxFlow with the unrolling level being the respective loop bound, and set the allowed deviation to 0.1 bits and a correctness probability of 0.95. Both the lower allowed deviation and the higher correctness probability increase the run-time and the precision and result in different values than the default configuration for the same unrolling levels.

Results. The computed leakages are given in Table 1 and show that Nildumu over-approximates the leakage for every program and every level of inlining, in contrast to ApproxFlow which under-approximates the leakage if the unrolling and inlining level is lower than required by each program. Table 1 also shows that Nildumu has worse precision for most test cases involving arithmetic and comparison operators. Furthermore, Table 2 gives the execution time for all programs and shows that Nildumu is slower than ApproxFlow. Additional benchmarks showed that Nildumu does not produce better results, performance and leakage-wise, when using the PMSAT based leakage computation with Open-WBO.

Discussion. ApproxFlow is by design more precise for programs where it can fully unroll all loops and inline all functions, as it models operators directly as a SAT formula. Nildumu only uses simple dependencies between bits and not complex, SAT-based dependencies as ApproxFlow and has, therefore, worse precision, especially for arithmetic operators. Nonetheless, Nildumu analyses the presented benchmark programs with comparable precision and gives an over-approximation for every benchmark and unrolling limit. The results also show that using Nildumu without support for path conditions leads to worse precision with performance gains for only part of the benchmarks.

The run-time of Nildumu is worse than the run-time of ApproxFlow. This is partly due to its implementation in Java, compared to ApproxFlow which is a small Python wrapper combining two tools written in C++, and due to the additional computation of summary graphs which is especially expensive as this computation over-approximates the effect of all remaining recursion.

8 Conclusion and Future Work

In this paper, we presented a QIF analysis exploiting bit dependency graphs that supports a while-language with loops, recursive functions, fixed-width integers, and fixed-size arrays. To our knowledge, this is the first analysis that supports recursion (and loops) without a limit on the recursion depth using summary graphs as an adaptation of the well-known concept of summary edges. This reduces the set of assumptions on the processed programs. The analysis computes an upper bound of the information leakage using min-entropy regardless of the level of inlining.

The evaluation results presented in Sect. 7 show that the analysis produces comparably good results for typical examples, but also that the arithmetic and comparison expressions are conservatively approximated, and that the performance and precision could be improved. Especially the construction of summary graphs and the handling of arrays should be improved to reduce the execution time of the analysis. The analysis could therein profit from parallelization.

Furthermore, the used summary graphs are currently limited as their construction ignores the specific call-sites and their context. This problem should be addressed in future extensions of this approach, for example by the techniques already developed for data flow analyses in compilers. There is ongoing work to support a broader range of language features (like input and output streams) as well as using CBMC as a front-end to improve the real-world applicability of Nildumu. The precision could be improved by using interval-based lattices or incorporating more operator semantics using techniques from the field of bounded model checking. Furthermore, the analysis could be extended into a component-based analysis which analyzes program components and the flows between them.

Acknowledgements. This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89), and by the German Federal Ministry of Education and Research within the framework of the project KASTEL SVI in the Competence Center for Applied Security Technology (KASTEL).

References

1. Assaf, M., Signoles, J., Totel, E., Tronel, F.: The cardinal abstraction for quantitative information flow. In: Workshop on Foundations of Computer Security 2016 (FCS 2016), Lisbon, Portugal (June 2016). https://hal.inria.fr/hal-01334604
2. Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: 2009 30th IEEE Symposium on Security and Privacy, SP 2009, pp. 141–153. IEEE, Washington, DC (May 2009). https://doi.org/10.1109/SP.2009.18

3. Beyer, D., Gulwani, S., Schmidt, D.A.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_16
4. Biondi, F., Enescu, M.A., Heuser, A., Legay, A., Meel, K.S., Quilbeuf, J.: Scalable approximation of quantitative information flow in programs. In: VMCAI 2018. LNCS, vol. 10747, pp. 71–93. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_4
5. Biondi, F., Legay, A., Quilbeuf, J.: Comparative analysis of leakage tools on scalable case studies. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 263–281. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_17
6. Bondy, J.A., Murty, U.S.R.: Graph Theory. Graduate Texts in Mathematics, Springer, Heidelberg (2008). https://doi.org/10.1007/978-1-84628-970-5
7. Budiu, M., Sakr, M., Walker, K., Goldstein, S.C.: BitValue inference: detecting and exploiting narrow bitwidth computations. In: Bode, A., Ludwig, T., Karl, W., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 969–979. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44520-X_137
8. Cha, B., Iwama, K., Kambayashi, Y., Miyazaki, S.: 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 1997/IAAI 1997, pp. 263–268. AAAI Press (1997)
9. Chadha, R., Mathur, U., Schwoon, S.: Computing information flow using symbolic model-checking. In: Leibniz International Proceedings in Informatics, LIPIcs, vol. 29, pp. 505–516 (2014). https://doi.org/10.4230/LIPIcs.FSTTCS.2014.505
10. Cherubin, G., Chatzikokolakis, K., Palamidessi, C.: F-BLEAU: fast black-box leakage estimation. In: Proceedings - IEEE Symposium on Security and Privacy 2019, pp. 835–852 (May 2019). https://doi.org/10.1109/SP.2019.00073
11. Chothia, T., Kawamoto, Y., Novakovic, C.: LeakWatch: estimating information leakage from java programs. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014, Part II. LNCS, vol. 8713, pp. 219–236. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11212-1_13
12. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. J. Comput. Secur. **15**(3), 321–371 (2007). https://doi.org/10.3233/JCS-2007-15302
13. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
14. Esfahanian, A.H.: Connectivity algorithms (2013)
15. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. In: Paul, M., Robinet, B. (eds.) Programming 1984. LNCS, vol. 167, pp. 125–132. Springer, Heidelberg (1984). https://doi.org/10.1007/3-540-12925-1_33
16. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. (TOPLAS) **12**(1), 26–60 (1990). https://doi.org/10.1145/989393.989419
17. Klebanov, V.: Precise quantitative information flow analysis - a symbolic approach. Theor. Comput. Sci. **538**, 124–139 (2014). https://doi.org/10.1016/j.tcs.2014.04.022
18. Malacaria, P.: Assessing security threats of looping constructs. In: Conference Record of the Annual ACM Symposium on Principles of Programming Languages, pp. 225–235 (2007). https://doi.org/10.1145/1190216.1190251

19. Mantel, H.: Information flow control and applications—bridging a gap—. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 153–172. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45251-6_9
20. Martins, R., Manquinho, V., Lynce, I.: Open-WBO: a modular MaxSAT solver. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 438–445. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_33
21. McCamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), vol. 43 (2008)
22. Meng, Z., Smith, G.: 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 2011, pp. 1:1–1:12. ACM, New York (2011). https://doi.org/10.1145/2166956.2166957
23. Mu, C.: Computational program dependence graph and its application to information flow security. Newcastle University, Computing Science (2011)
24. Muchnick, S.: Advanced Compiler Design Implementation. Morgan Kaufmann, Burlington (1997)
25. Newsome, J., McCamant, S., Song, D.: Measuring channel capacity to distinguish undue influence. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, PLAS 2009, pp. 73–85. ACM, New York (2009). https://doi.org/10.1145/1554339.1554349
26. Phan, Q.S., Malacaria, P., Tkachuk, O., Păsăreanu, C.S.: Symbolic quantitative information flow. SIGSOFT Softw. Eng. Notes **37**(6), 1–5 (2012). https://doi.org/10.1145/2382756.2382791
27. Smith, G.: Recent developments in quantitative information flow (invited tutorial). In: 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 23–31 (July 2015). https://doi.org/10.1109/LICS.2015.13
28. Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2009), vol. 5504, pp. 288–302. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00596-1
29. Wegbreit, B.: Property extraction in well-founded property sets. IEEE Trans. Softw. Eng. SE **1**(3), 270–285 (1975). https://doi.org/10.1109/TSE.1975.6312852