Precisely Quantifying Software Information Flow

by

Mihai Adrian Enescu

B.Sc., The University of British Columbia, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

March 2016

Abstract

A common attack point in a program is the input exposed to the user. The adversary crafts a malicious input that alters some internal state of the program, in order to acquire sensitive data, or gain control of the program's execution.

One can say that the input exerts a degree of *influence* over specific program outputs. Although a low degree of influence does not guarantee the program's resistance to attacks, previous work has argued that a greater degree of influence tends to provide an adversary with an easier avenue of attack, indicating a potential security vulnerability.

Quantitative information flow is a framework that has been used to detect a class of security flaws in programs, by measuring an attacker's influence. Programs may be considered as communication channels between program inputs and outputs, and information-theoretic definitions of information leakage may be used in order to measure the degree of influence which a program's inputs can have over its outputs, if the inputs are allowed to vary. Unfortunately, the precise information flow measured by this definition is difficult to compute, and prior work has sacrificed precision, scalability, and/or automation.

In this thesis, I show how to compute this information flow (specifically, channel capacity) in a highly precise and automatic manner, and scale to much larger programs than previously possible. I present a tool, nsqflow, that is built on recent advances in symbolic execution and SAT solving. I use this tool to discover two previously-unknown buffer overflows. Experimentally, I demonstrate that this approach can scale to over 10K lines of real C code, including code that is typically difficult for program analysis tools to analyze, such as code using pointers.

Preface

The work I describe in this thesis is mostly my own, with the exception of the subSAT solver which was designed by Sam Bayless.

I have based much of the text on a paper due to appear and to be presented at the European Symposium on Security and Privacy 2016, titled "Precisely Measuring Quantitative Information Flow: 10K Lines of Code and Beyond", which was written in collaboration with Celina Val, Sam Bayless, Alan Hu, and William Aiello.

I have based some of the writing on previous (but unpublished) submissions to the USENIX Security Symposium 2015, and the IEEE Symposium on Security and Privacy 2015, with the same co-authors.

Table of Contents

List of Tables

List of Figures

Acknowledgments

First, I wish to express my gratitude to my advisors, Professors Alan J. Hu and William Aiello, for their unyielding guidance and support, and seemingly infinite patience throughout my time in grad school.

Moreover, I'm grateful to the members of the Integrated Systems Design laboratory, who always provided me with great ideas and feedback and listened to me drone on about my work on many occasions. In particular, I extend a special thanks to Celina G. Val and Sam Bayless, for their generosity in donating countless hours and ideas, and their respective projects Kite and ModSAT, without which nsqflow, and this thesis, would not have been possible.

I would also like to express my thanks to Professor Mark Greenstreet, whose encouragement and advice played an integral role in my decision to attend graduate school, and who has offered excellent feedback on this work on multiple occasions.

Dedication

I dedicate this work to Emilia, Gabriel, and Cornel Enescu, without whose unconditional love and support I would not have completed this thesis.

Chapter 1

Introduction

In this thesis, I address the problem of measuring the degree of influence of a program's inputs over its outputs. I first motivate this problem by providing examples of its economic cost. I then introduce previous related problems and outline their existing solutions, providing examples that demonstrate their shortcomings, demonstrating the importance of the problem addressed by this thesis.

In a vulnerable software system, a malicious adversary can construct a user input that triggers a bug in an executing program that will either grant the adversary control over program execution, or allow the adversary to extract sensitive information.

SQL injection attacks are a well-known example where the adversary exploits errors in the code to illicitly obtain sensitive information [11]. In 2014, the NTT Group estimated that an organization can incur costs of $\$196,000$ for a single SQL injection attack [14]. Such attacks are commonplace, and a list of well-known attacks since 2002 is maintained by Hicken [13], who lists attacks such as the 2011 theft of 7 million private customer details from the Sony PlayStation Network, and the 2012 theft of account details for 1.6 million records belonging to U.S. government agencies. Furthermore, Hicken also lists many smaller attacks which obtained credentials used to gain control of web servers and other systems.

However, the crafting of malicious input is not limited to SQL injection. Indeed, the economic and social impact of malicious input attacks far exceeds the numbers I have just presented.

Such a malicious input attack can be described as follows. By varying the public program input, an adversary exerts a degree of *influence* on a specific program output. One of the oldest methods of quantifying influence is *taint analysis*, which has long been used[1] to determine whether untrusted input data can inadvertently influence the value of trusted or sensitive data. Values in a program that are directly computed on the basis of the untrusted

[1] The earliest published use of the term that I have found is the original Perl book [40]. Schwartz et al. [33] provide a fairly recent survey of the large literature on taint analysis and its applications.

```
function InputSanitizer1(input)
    if input is in legal range then
        return input
    else
        return ERROR

function InputSanitizer2(input)
    return input
```

Figure 1.1: This simple example contrasts a stylized input sanitizer with a function that simply passes the input unchecked. Taint analysis does not distinguish between the two.

input are considered "tainted". Subsequently, a taint policy determines how taint is propagated dynamically through runtime operations. As a very coarse, conservative abstraction, taint analysis is efficient but suffers from excessive false positives leading to an overestimate of the influence of untrusted inputs on sensitive data. Taint analysis also typically addresses the problem of *confidentiality* (detecting sensitive data leaks), rather than program *integrity* (measuring the degree of an adversary's control).

For a simple example, consider the two functions in Figure 1.1. The first is a stylized input sanitizer. The second skips all checks. Evidently, the former is correct and the latter is not, but taint analysis would label both functions' return values equally tainted. Furthermore, as typically implemented, taint analysis can produce false negatives (underestimates of leaked information) as well. These result from a desire to improve efficiency (e.g., a dynamic analysis that considers only a small number of program paths) or reduce false positives (e.g., by ignoring implicit information flow when tainted information affects control flow). Newsome et al. [25] give several real-life examples (drawn from common Unix utilities, the Windows keyboard driver, and how GCC compiles switch statements) of coding patterns that cause typical taint analyses to produce both false positives and false negatives.

A *quantitative* measure of information flow is an elegant solution to the shortcomings of taint analysis, and considerable research has pursued this direction. Returning to Fig. 1.1, two reasonable questions to ask would be whether or not InputSanitizer2 gives the input full control of the return value, and whether or not InputSanitizer1 limits the influence to $\log_2(|\text{legal range}| + 1)$ bits of information.

In this thesis, I am primarily concerned with program integrity. At a high level, the goal of program integrity is to ensure that inputs from untrusted sources cannot cause a system to perform unsafe actions, according to some specification of safety. This work, in particular, was inspired by Newsome et al. [25], whose goal was to statically compute a measure for program integrity. They define the *influence* that a (potentially untrusted) program input can have over a specified program variable in two steps. First, they consider the number of distinct values of the specified variable that the program can be forced to compute by ranging over all possible input values. They define the bits of influence of the input over the specified variable as the $\log_2$ of this number. For example, if the specified variable is 32 bits and is sensitive, a large number of bits of influence should flag a potential problem.

As stated, this definition of influence given by Newsome et al. [25] is equivalent to the channel capacity. The number of bits of influence is equal to the maximum number of bits of uncertainty of the output value, over all possible input distributions. Even without the connection to information theory, influence is a natural worst-case notion in a security context for capturing in a single value the amount of control an adversary may have over a specified program variable. In Chapter 6, I consider a generalization of the worst-case role that auxiliary inputs (i.e., not under the adversary's control) can have on the adversary's influence on a specified program variable.

In this thesis, I address the same influence computation given by Newsome et al.: given a deterministic program, an exit point where information flow is measured, and a specified set of output variables, compute the cardinality of the set of all possible values of the output variables after all executions from the program's inputs to its outputs at the exit point. I treat all inputs as non-deterministic, e.g., under the control of the attacker. My goal is to compute this cardinality precisely.

Prior work has sacrificed scalability, precision, and/or automation. Most publications that attempt precise (or close approximations of) channel capacity give results only for small examples of around a dozen lines of code or less (e.g., [2, 22, 25, 27]). On the other hand, Newsome, McCamant, and Song [25] also report impressive scalability in analyzing known vulnerabilities on real systems code (RPC DCOM, SQL Server, ATPhttpd), but these results are based on analyzing only a single execution trace (thereby ignoring many possible information flows), and on these examples, their computation yielded lower bounds of fractionally above 6 bits and upper bounds of 32

bits or slightly less for a 32-bit output. The lower bounds were sufficient to suggest the presence of an attack, but these are clearly not precise information flow bounds. My work was directly inspired by theirs — my goal is also to handle real system code, but with much greater precision on the channel capacity measurement. Similarly, McCamant and Ernst [21] presented an analysis that scaled to over 500KLoC, but based on rough, conservative tracking of the bit-widths (not information content) of data flows and some manual program annotations. Most similar to my work is a recent paper by Phan and Malacaria [26], who compute precise quantitative information flow and report scaling on three examples to over 200 lines of code. Like them, I am also using symbolic execution and modern SAT/SMT techniques. As I will show in Section 4.3, nsqflow is far more scalable.

Contributions:

- I demonstrate for the first time that highly precise quantitative information flow computation can scale to real, medium-sized programs. Specifically, I present *nsqflow*, an automated static analysis tool for C source code that measures the channel capacity between a program's inputs and user-selected outputs. *nsqflow* is both precise (subject to the limitations below), and able to scale to 20K lines of real C code. In my experiments, nsqflow flagged two previously-undisclosed buffer overflows in real, open-source programs.
- To facilitate the analysis of real C source, I make extensions to Kite, a state-of-the-art symbolic execution tool, in order to efficiently analyze code with pointers.

Chapter 2

Background

2.1 Quantitative Information Flow

Quantitative information flow (QIF) is a framework that has been used in detecting a class of security flaws in programs. In QIF, programs are considered communication channels between inputs and outputs, and information-theoretic definitions of information leakage may be used in order to measure the degree of influence which a program's inputs can have over its outputs, if the inputs are allowed to vary.

Not surprisingly, this is a difficult problem. Intuitively, there is the challenge of analyzing an explosion of possible program paths to the specified exit points, as well as the #P-complete problem of enumerating satisfying assignments that generate possible output values. Formally, Černý, Chatterjee, and Henzinger [39] show that merely bounding the cardinality is PSPACE-complete. Furthermore, this complexity is in terms of the explicit state space of a finite-state program (the set of all possible valuations of all possible program states), not just in terms of program size. Even in the simple case of loop-free Boolean programs, Yasuoka and Terauchi [42] show that the problem is coNP-complete.

Sabelfeld and Myers [31] provide an extensive survey of earlier work on information flow, and Smith [35] provide a full treatment of the theoretical foundations of quantitative information flow in programs.

Moreover, there has been work in practically measuring QIF. In [2], the authors show how to compute a precise measure of information flow for small programs for a variety of information-theoretic measures by counting the number of equivalence classes in an equivalence relation over program variables. However, the tool they present scales only to small programs on the order of a dozen lines.

Meng and Smith [22] present a way to compute Renyi's min-entropy by analyzing relations between pairs of bits in a program's outputs, but only present results for small programs on the order of 10-20 lines of code (including many of the smaller examples from [25]), and show a rapid exponential blowup as they scale up the number of program paths (15 hours for 2^{32}

program paths).

Newsome, McCamant, and Song [25] define a program input's *influence* over the program output as the the $\log_2$ logarithm of the number of solutions to the outputs, given that the inputs are allowed to vary. They argue that if the influence is large relative to the output width (for instance 32 bits for a 32-bit output), this is somewhere that a problem is likely to be present. Their measurement is equivalent to *channel capacity*, which represents a worst-case over all possible distributions of the input variables, of the number of bits of uncertainty. Their approach, based on symbolic execution, is able to scale to large programs at the cost of precision. A more complete comparison of nsqflow with [25] is presented in Section 5.2

Phan and Malacaria [26] compute precise QIF based on a symbolic execution engine that counts the number of solutions while performing symbolic executing, in order to arrive at a precise measurement. They scale to programs on the order of 200 lines of code. A more complete comparison of nsqflow with [26] is presented in Section 5.2.

2.2 Kite: Conflict-Driven Symbolic Execution

I implement nsqflow as an extension to Kite, a tool that implements a state-of-the-art symbolic execution algorithm known as *conflict-driven symbolic execution* [37]. The reader is assumed to be familiar with symbolic execution and SAT solvers based on conflict-driven clause learning (CDCL). Please refer to [3, 12, 15] for a fuller treatment of symbolic execution, and [1, 9, 24] for background on CDCL SAT solvers.

A symbolic execution engine operates on symbolic values (sets of concrete values), executing instructions sequentially in order to prove or disprove a property about the program (typically assertion statements). Every time there is a branch point in the program, the symbolic execution must consider (but not necessarily explore) both program paths. Symbolic execution will simulate a fork in the program execution. If a branch is infeasible, the symbolic execution does not need to explore this path, and will continue executing only along feasible paths. Eventually, all paths in the program will be covered as long as the path length is finite.

Conflict-driven symbolic execution (CDSE) [37], is a symbolic execution algorithm that is able to learn from infeasible paths. CDSE initially encodes the program's control flow graph (CFG) as a CNF formula (in a separate formula from the path constraints). It then augments this formula with further constraints each time a path is proven to be infeasible by the path

constraint solver, to exclude the infeasible path, along with potentially many others, from the solution space of the CFG formula.

```
1    int main () {
2        symbolic int x, y;
3        if (x < 0)
4            x = -x;
5        int result = x - y;
6        if (x < y)
7            result = y - x;
8        assert (result >= 0);
9    }
```

Listing 2.1: An example (adapted from Listing 1.1 of [37]) to illustrate that not all paths are needed to prove a property. In this example, the property to be proven is an assertion statement embedded in the code. In this example, this assertion always holds.

For a demonstration of path learning in Kite, consider the small program given in Listing 2.1. A corresponding execution tree encoding each program statement and branch as a node is shown in Figure 2.1. The transitions following each node are annotated with the full path condition that Kite has learned following the program statement. Although there are 4 distinct execution paths that reach the target assertion, Kite need not explore each distinct path, unlike traditional symbolic execution engines. For the case when Kite evaluates the $(x < y)$ branch to True, the conjuncts in the path condition that are shown in red, specifically $(x < y)$ and (result$= y - x$) are sufficient to prove the assertion. These conjuncts correspond to lines 6 and 7 in the code. After Kite has explored the first (leftmost) path, it need not evaluate the assertion again in the third path from the left, as it has learned a clause that any time these conjuncts are present, the assertion will evaluate to false. Similar reasoning can be applied to the second and fourth paths, and Kite only needs to fully explore 2 of the 4 program paths in order to prove the assertion. Specifically, Kite will exclude paths that reach the assertion in Line 8 with `!(result >= 0)` in the path condition. Any future paths that have this clause in the path condition will be refuted. As a result, Kite need not explore all the paths through the program.

Each time that a path constraint is proven to be unsatisfiable, it follows that at least one different branch in the path must be taken in order to reach the output of the program, and in principle, it would be sufficient to refine the CFG formula with this clause. However, on unsatisfiable instances,

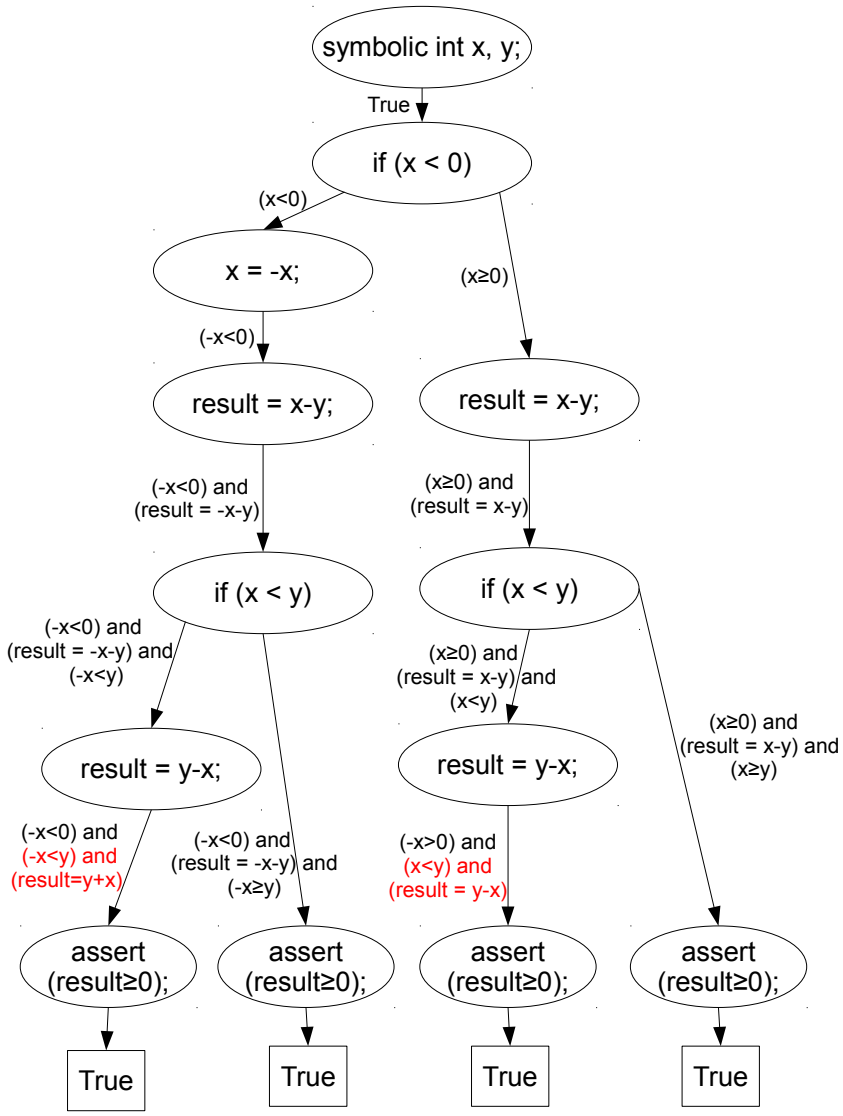

Figure 2.1: The full execution tree corresponding to the code in Listing 2.1. The path condition following each program statement is labeled on the transition following each statement. Each path condition is expressed in terms of the initial values of the variables appearing in the path condition (the initial symbolic values given in line 2). I have shown path conditions as linear arithmetic constraints for ease of reading, but it should noted that such constraints would need an SMT solver — not just a SAT solver.

modern SAT and SMT solvers can derive a *conflict clause*, which in this case will describe a sufficient subset of the branches in the infeasible path such that at least one of the branches in that subset must differ in any feasible path to the output — and in many cases, this subset may be much smaller than the full set of branches in the path.

Analogously to clause learning in a CDCL SAT solver, after each infeasible path, Kite adds these conflict clauses to the CFG formula, potentially blocking many previously unexplored (but provably infeasible) paths, without giving up completeness. Each satisfying solution to this CFG formula describes a potentially feasible path in the program; decisions on which path to explore next are made by repeatedly finding satisfying solutions to this formula.

By maintaining this CFG formula and using it to direct the sequence of path explorations, CDSE takes advantage of two features responsible for the recent success of CDCL SAT solvers: conflict analysis and non-chronological backtracking. This allows Kite to perform complete symbolic execution while mitigating the path explosion problem.

Chapter 3

nsqflow

I have developed a tool, nsqflow, to measure QIF (specifically, channel capacity) through programs. In this chapter I describe the architecture of my tool. Figure 3.1 presents a high-level overview of my approach, showing the various components of the toolchain.

As part of this process, the user must select a variable along with a location in the code over which they wish to measure influence. Currently, I achieve this with a simple annotation, but in principle this could be applied to unaltered code, for example using a GUI to select the variable in question.

This program is symbolically executed using the symbolic execution engine Kite [37] to produce a representation of the program as a Boolean Satisfiability (SAT) instance in conjunctive normal form (CNF). During this step, Kite forms a *path condition* for each path that it explores, and employs a constraint solver to both simplify and solve each path condition. I describe this procedure more closely in Section 3.1.

In addition to being solved, nsqflow stores the CNF representations of each path condition encountered during symbolic execution. Finally, once all paths have been explored, the CNFs for each path condition are combined and handed to a model counter, which counts the number of distinct[2] satisfying assignments to *just the output variables* in the resulting CNF. Finally, the reported measurement is the base-2 logarithm of the number of such satisfying assignments.

In the following sections, I describe the various components of nsqflow.

3.1 Architecture of nsqflow

I have implemented *nsqflow* using the symbolic execution engine Kite, which implements the CDSE algorithm [37] and was developed primarily to verify embedded assertions in sequential programs. Consequently, nsqflow inherits many components from Kite. Kite verifies programs described in the LLVM

[2] By distinct, I mean that any set of solutions that share the exact same values on the output variables are counted only once.

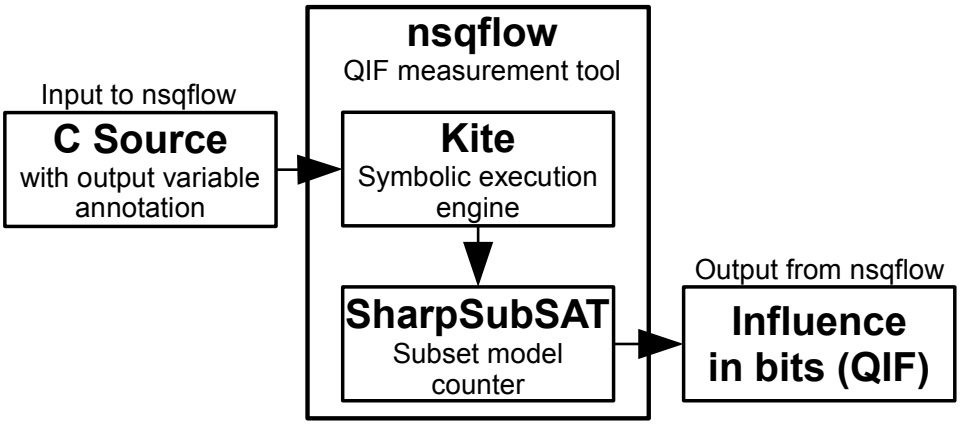

Figure 3.1: The architecture of nsqflow. Note that Kite outputs a CNF formula encoding the feasible paths of the source program. This CNF is fed as input to SharpSubSAT to compute the subset model count. The influence is the base-2 logarithm of this count.

assembly language [18], known as LLVM-IR, which can be obtained by compiling C files using the LLVM C front-end. nsqflow has inherited LLVM as its underlying compiler suite, and internally uses LLVM-IR during its computation of QIF. nsqflow has also inherited Kite's internal constraint (SMT) solver and SAT solver (STP [3], and MiniSat 2 [9], respectively).

Kite is able to generate CNFs representing the set of all feasible paths leading to a desired target property, and nsqflow leverages this feature in generating a final CNF to be model-counted. Unfortunately, Kite does not implement pointer analysis, treating symbolic pointers conservatively. This leads to unacceptable running times for pointer-heavy C code, and I discuss how I address with this issue in section 3.1.1, adding pointer analysis to Kite.

To achieve a precise measure of information flow, nsqflow symbolically executes all the relevant (feasible) paths to the output variables of the (finite) program being analyzed, and then forms the disjunction of those paths into a single CNF for model counting (as described below). However, naively attempting to symbolically execute *all* program paths scales very poorly in practice, as the number of paths to execute may be exponential in the number of branches — and many of those paths might be irrelevant to the information flow currently being analyzed.

In order to support model counting in this paper, I implement as part of

nsqflow two further changes to Kite: First, nsqflow adds a blocking clause to the CFG formula each time a feasible path to the output variable is found, to prevent that complete path from being explored again (this forces Kite to continue symbolically executing all feasible paths to the output, rather than just halting after the first path is found). Secondly, nsqflow stores the formulas for each feasible path constraint, and forms their disjunction as one monolithic output CNF to be passed to the subset model counter (described below). Only the *feasible* path constraints must be included in this disjunction; infeasible path conditions need not be stored, reducing the size and difficulty of the resulting CNF.

During symbolic execution, the accumulated path constraints from root to leaf in the program's execution tree (for feasible paths) represent the branch decisions leading from specific concrete inputs to the output value. A disjunction of the constraints over all feasible paths represents a full formula for a finite program. While the number of paths is exponentially large, Kite is able to prune large parts of the search space and make possible an incrementally-updated disjunction.

The basic implementation details of forming the incremental path disjunction are as follows. Initially, nsqflow requires a user-provided *flow declaration* as a way to mark the output variables at a specific point in the program. This is implemented as a special function call in the program: `DECLARE_OUTPUT(varname)`. Conceptually, this function call is translated into a pair of assertions in the code: `assert(varname != 0)` and `assert(varname == 0)`, which comprise the target property for the symbolic execution engine to check. To support an incremental update to the CNF, nsqflow keeps track of program-variable-to-STP, and STP-to-SAT variable mappings, performing CNF variable renaming to avoid any name collisions. nsqflow also imposes equality constraints on the CNF variables corresponding to the same program variable across different paths, in order to keep the CNF correctly constrained in the number of admitted solutions. The implementation is memory-efficient, in that at any given time, nsqflow need maintain only these constraints for the previous path and the current path.

3.1.1 Memory Handling

Due to Kite's lack of pointer analysis, it would be impossible to analyze real C code containing pointers. Because Kite assumes that all pointers are 32 bits, Kite branches on all 2^{32} possible memory addresses when there

is a symbolic pointer dereference. While this is a sound upper bound on the number of feasible execution paths, it is an infeasible approach. Consequently, I have implemented a method to better handle memory access, a facility necessary for nsqflow to scale to real code. Below, I describe how symbolic pointers are handled in nsqflow.

Every time Kite makes a decision on a CFG branch, nsqflow makes a query to the pointer analysis provided by LLVM (called Data Structure Analysis (DSA) [19]) under the current path condition, in order to learn aliasing information. For pointers that DSA finds to be definitely equal or definitely not equal, nsqflow extends the path condition at the time of the previous CFG branch decision with this information. Thus, further exploration extending that path condition will be augmented with the returned pointer comparison (aliasing) information.

DSA itself works by first constructing a directed graph for each function in the program, with each graph consisting of a set of nodes that correspond to the memory objects named in the function (stack, heap, and global objects allocated or named in the function, including basic integer and floating-point types, arrays, structures, pointers, and functions), edges between the fields of nodes when corresponding objects name each other in the program, a set of edges mapping these names to the fields they name, and a set of all function calls made by the function. This information is enough for DSA to simplify the graph, removing duplicate call sites when calls are made to the same function with the same arguments and return types. Once DSA has finished constructing directed graphs for each function in the program using local information as just described, it copies the local graphs of each callee into the calling function's local graph, detecting strongly-connected components (SCCs) in order to avoid visiting each SCC more than once (in order to avoid infinite cycles), eliminating duplicate calls in the same manner as during the local graph construction. The result of this interprocedural copying is a full call graph for the program. Finally, each graph is traversed again, with the calling function's graph being copied into the graph of each potential callee. In this manner, DSA is able to scale efficiently, requiring only seconds for programs spanning hundreds of thousands of lines of code, with a worst-case time complexity of $\Theta(n\alpha(n) + k\alpha(k)e)$ and a worst-case space complexity of $\Theta(fk)$, for a program with n instructions, k as the maximum size of a graph for a single function, e edges in the program's call graph, f functions in the program, and where α is the inverse Ackermann function. Due to this low computational overhead and excellent scalability, nsqflow is able to make a pointer query every time Kite makes a CFG branch decision, and explores a new path under the path constraint at the time of

Kite's branch decision.

In addition to being an alias analysis, DSA is a points-to set memory analysis [7, 19]. Unlike alias analysis, which returns equality and disequality relations between pairs of pointer variables, a points-to analysis for a pointer variable will yield a set containing memory objects which each pointer may reference. I extend Kite to use this information as part of its search decisions when the result of a conditional expression depends on a symbolic pointer. When the points-to analysis is successful, nsqflow need only consider a number of branches corresponding to the memory locations from the points-to set, instead of the potential 2^{32} memory addresses. The astute reader will observe that even this conservative treatment of symbolic pointers when unable to learn anything from DSA does not affect the precision of the resulting influence measurement. As the symbolic execution engine considers each possible memory address assigned to the pointer, it will determine that the paths along memory addresses which cannot be assigned are infeasible, and therefore cannot reach the target property. Thus, although it can have a dramatic impact on performance, it does not sacrifice precision.

One drawback of this alias analysis approach, however, is the increase in the size of the resulting CNFs by inserting clauses. Although these clauses provide additional constraints, which intuitively should make the intermittent SAT solving (as well as the final model counting) easier, it is not guaranteed to do so and can in some cases increase the difficulty of the problem given to the solvers.

3.1.2 Library Calls

A limitation of nsqflow is one that static analysis tools fundamentally suffer — that of library and system calls, or to functions for which we possess headers, but not the source implementation. Although nsqflow is a source analysis tool, real C code makes use of included header files for which the source is not available, and makes heavy use of system calls (in my experiments in chapter 4, the code analyzed makes heavy use of network reads, for instance). The default way in which nsqflow deals with the complexity of calls to such code is to implement stub libraries marking these functions as behaving *nondeterministically*, and potentially returning any value. Kite provides the facility to include these "fake" library calls. Furthermore, for commonly used functions, the user of the tool can implement *models* for the functions to account for the behavior of these calls. A model will help the symbolic execution engine prune many paths from its search, but if the model is incorrect, precision will also suffer.

3.1.3 Relaxing Precision

Although my goal is to measure influence precisely, one advantage of nsqflow's incremental, path-by-path nature is that it facilitates the early termination of its execution to obtain a lower bound on the true channel capacity. This may be useful to implement policies such as "the inputs should have no more than m bits of influence over the output". This formulation can be useful to verify programs for which a low influence measurement is expected (for instance, string sanitizers on untrusted user inputs) — a large output set may be a sign of an integrity violation. For instance, consider a program that is intended to output a small set of values, but nsqflow discovers a large (but not exhaustive) number of solutions to the output set, such as a buggy piece of code that is expected to return one of only 3 or 4 possible values (e.g. return codes) for an output of width 32 bits. In this case, observing a measurement of 10 bits (for instance), is enough to alert the user to the issue, without requiring a full exploration of all 2^{32} possible outputs.

There are two potential points at which nsqflow may be terminated early to obtain this bound. In the first, symbolic execution may be stopped early to obtain a CNF that represents only a subset of the paths explored at that point. Because nsqflow maintains a CNF that is incrementally-updated as Kite explores new paths, terminating symbolic execution early will yield a CNF with a number of solutions that is a sound lower bound on the total number of solutions. Thus, model counting this CNF will yield a sound lower bound on the influence measurement.

In the second, the model counting step may also be terminated early. Due to the incremental nature of the model counter used in nsqflow, it can be stopped at any point to obtain a lower bound on the total number of solutions to the CNF.

When exact precision is not required, as in the case of the aforementioned minimum-threshold policy, the savings in running time can be dramatic. Indeed, early termination of either of these steps may lead to an exponentially lower number of paths to consider for the symbolic execution engine.

3.2 Subset Model Counter

In the final step of my approach, nsqflow counts the number of different satisfying assignments that the output variables can be assigned in the CNF representation of the program, which corresponds to the size of the feasible set of values of the output variables. In order to compute the size of the

feasible value set for just the output variables (as opposed to the potentially much larger count of all possible satisfying assignments to the CNF, which may include many assignments which differ only in other variables), I use a variant of model counting, which I call *subset model counting*, or (#SubSAT), as the basis for computing the size of the feasible value set. Whereas in exact model counting, one must count the number of unique satisfying assignments to CNF formula ϕ, in #SubSAT one must count the number of unique assignments to a subset S of the variables of ϕ that can be extended into satisfying assignments of ϕ.

A typical CDCL-based model counter [32] can be easily adapted to subset model counting by simply removing all literals from the blocking clause that are not elements of the subset S. Such an algorithm is described in Algorithm 3.1. Correctness of this algorithm follows from the correctness of the very similar 2QBF algorithm (Alg. 1) described in [29].

Algorithm 3.1 SharpSubSAT Model Counter

1: $S \subset vars(\phi)$ is the subset of Boolean variables in ϕ to count.
2: num_models = 0
3: **while** ϕ is SAT **do**
4: Let $assign$ be a satisfying assignment to ϕ.
5: Let $assign' \subset assign$ be a prime implicant of ϕ.
6: $assign'_S \leftarrow \{x \mid x \in assign' \wedge var(x) \in S\}$.
7: num_models $\leftarrow$ num_models $+ 2^{|S|-|assign'_S|}$
8: $\phi \leftarrow \phi \wedge \neg assign'_S$

I implement this step in the nsqflow toolchain using the #SubSAT solver SharpSubSAT [38], a prime-implicant-based CDCL model counter [4, 8, 20, 23] that uses a greedy cover-set approach to deriving prime implicants. SharpSubSAT implements Algorithm 3.1 with the heuristics of first deciding on the variables of S; and adding variables *not* from S to the cover set first (thus potentially reducing the number of literals in S that end up in the prime implicant and subsequent blocking clause). In my experiments, solving the subset model counting problem (for small subsets of up to a few hundred variables) is dramatically faster than complete exact model-counting, and is an important heuristic for the scalability of the computationally hard model counting step of nsqflow. When SharpSubSAT has run to completion, the resulting count is precisely the cardinality of the feasible value set. Finally, nsqflow takes the base-2 logarithm of the cardinality in order to obtain the influence in bits.

3.3 Limitations

It is important to present the limitations of nsqflow, and of this approach to QIF.

First and foremost, nsqflow attempts to solve a computationally difficult problem. As nsqflow must fundamentally reason about an exponential number of paths to the program exit points, the number of satisfying assignments that yield possible possible output values is similarly large. As previously mentioned, multiple steps in the nsqflow toolchain belong to high complexity classes. As a result, the runtimes obtained from nsqflow are feasible on some subset of real programs, but in the general case, nsqflow may be unable to run to completion even in the case of finite path lengths. As presented, nsqflow is sensitive to the choice of program being analyzed.

Second, nsqflow computes only channel capacity. Since channel capacity represents the worst-case behaviour over input distributions, it can overestimate the degree of influence over the program outputs. This can lead to false positives – even when nsqflow computes a large number of possible assignments to a set of output variables, this does not necessarily indicate a problem with the code. Some computations will fundamentally, and correctly, exert a large degree of influence on their outputs. Such would be the case, for instance, in cryptographic algorithms based on bit substitution and permutation. In such a case, channel capacity would not be a useful measure in order to detect errors in the code. More generally, channel capacity is not always the most appropriate measure of information flow, when other measures such as Shannon entropy or Renyi's min-entropy may be preferable.

Furthermore, nsqfow inherits problems suffered by all symbolic execution tools. My analysis considers only finite-length execution paths. In particular, nsqflow has a configurable parameter to bound the loop unrolling; if the symbolic execution can exceed this bound, the tool warns the user.[3] In theory, nsqflow's bounds are always precise with respect to the set of all paths explored, are always guaranteed under-approximations with respect to all paths (i.e., considering extremely long and non-terminating paths as well), and can be made arbitrarily accurate by extending the symbolic execution runs to include increasingly longer paths.

In practice, the important challenge is scalability. As I will show in Chapter 4, nsqflow scales to much larger programs than previous ap-

[3] In my experimental results in Chapter 4, the bound is always set high enough to exhaustively explore all possible execution paths.

proaches [2, 22, 25, 26], and does so while maintaining a high level of precision. There remains a theoretical concern that the lower bounds provided by bounded symbolic simulations could underestimate the influence of a malicious input. However, given the state-of-the-art, I believe that nsqflow's emphasis on scalability over completeness is well-justified.

Moreover, nsqflow makes conservative abstractions about the environment. For example, nsqflow does not model the file system, so a read to a file is treated as non-deterministic (i.e., under an attacker's control). Similarly, system calls that are not explicitly modeled are treated as returning non-deterministic values. In a security context, these may be reasonable assumptions, but they can result in coarse overestimates of the adversary's influence.

Finally, my current implementation operates from the source-code level, so nsqflow cannot quantify information flow to machine-level artifacts, such as the program counter, as can be done e.g., by Newsome et al. [25]. In principle, my approach could perform analysis of binaries if provided with a front-end that could recover the CFG and functionality of the binary.

Chapter 4

Evaluation

Two key measures of performance for an information flow tool such as nsqflow are precision and scale. Previous work in the area has demonstrated a dichotomy between these two extremes, with the ideal being a tool that can measure flow precisely at scale. I evaluate where nsqflow lies along this scale-precision spectrum using a set of programs varying in size and complexity.

Because information flows through programs are difficult to reason about by a human, and there is a general lack of available precise flow measurement tools against which I can compare my own measurements, it is inherently difficult to validate the correctness of my tool. In order to achieve some validation of correctness, I present two sets of experiments in the following sections. In the first, I run my tool against a nontrivial program for which I can analyze the information flow precisely. In the second, I compare the measurements obtained by nsqflow to the measurements presented by Newsome, McCamant, and Song [25] and by Phan and Malacaria [26]. As I will demonstrate, nsqflow reports the same value whenever an exact value is provided in [25] and [26], and falls within the lower and upper bounds whenever an approximate value is presented in [25].

Finally in Section 4.4, I demonstrate that nsqflow is able to analyze programs at scale, by running it on a selection of open source software and reporting my results.

My reported running times take into account a full command-line invocation of the nsqflow toolchain, from the start of symbolic execution to the end of model counting. Since there is fixed overhead associated with this process (e.g. interprocess communication, internal compilation by LLVM/Kite, temporary file input/output), the examples that run to completion very quickly are unfairly penalized. Thus, the reported running times for the small programs are less meaningful and almost entirely an artifact of my implementation. However, this effect is negligible for longer-running experiments. I executed all experiments on an Intel Core i5-750 2.66GHz with 8MB of L2 cache and 4GB of RAM running 64-bit Ubuntu Linux.

4.1 Testing Correctness

In order to test the correctness of nsqflow, I analyze the output QIF of nsqflow on a nontrivial program for which the influence is known by construction. The program implements a family of universal hashing functions and is shown in Listing 4.1. It demonstrates nsqflow's precise information flow computation through both explicit data flow and implicit control flow.

```
int matrix[ROUNDS];
...
void fill_matrix(int seed) {
  int i;
  srand(seed);

  for (i=0; i < ROUNDS; i++)
    matrix[i] = rand() &
                ((1 << OUTPUT_WIDTH) - 1);
}
// print matrix to file
...
// matrix has constants hardcoded by fill_matrix
long hash(long x) {
  int i;
  int value = 0;
  for (i=0; i < ROUNDS; i++) {
    if (x & 0x1) value ^= matrix[i];
    x >>= 1;
  }
  return value;
}
```

Listing 4.1: Universal hashing example

The above generates a matrix of bits, with `ROUNDS` columns, each of which possesses `OUTPUT_WIDTH` bits by construction (masking the remaining bits in the bitvector to 0). The matrix is filled with deterministic values set by initializing the pseudorandom number generator with a particular value for its seed, resulting in the same sequence of pseudorandom numbers when `fill_matrix` is passed the same seed.

The hash function consumes one bit at a time from the value to hash

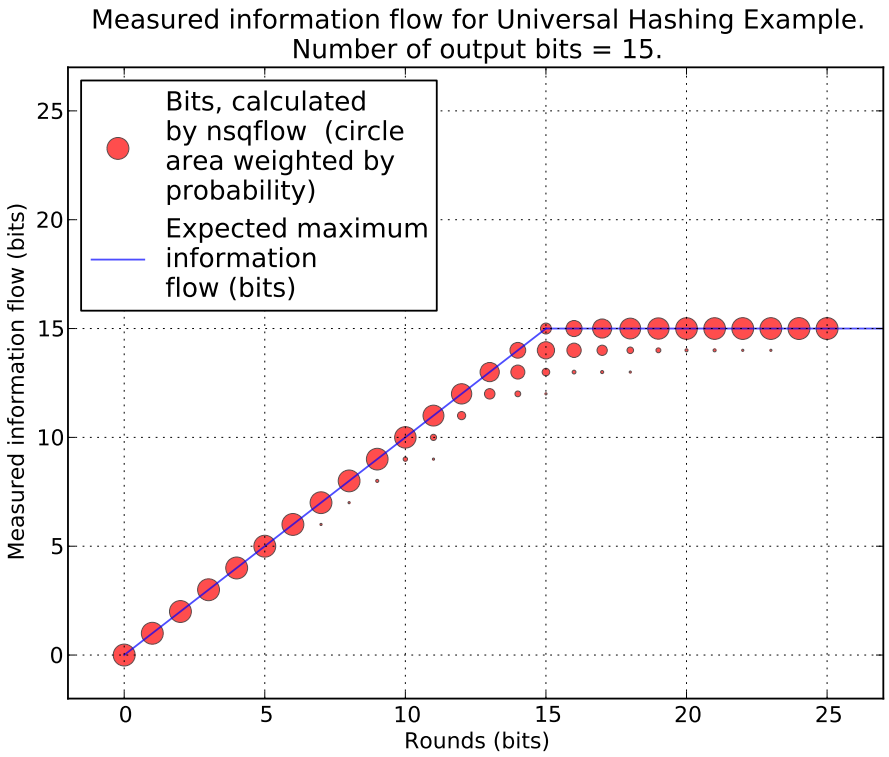

Figure 4.1: Universal hashing tests. The area of the points is proportional to the number of times that amount of information flow was measured, of my 100 runs.

(`x`), and if the bit's value is 1, it performs a bitwise xor with the bit in the next matrix column. In this way, one knows that by construction, at most `ROUNDS` bits of the return value may be affected by the input bits of `x`, dependent on the matrix values chosen. After a number of rounds (input bits) equal to `OUTPUT_WIDTH`, one expects to see a limit.

The only input in this case is `x`, and measurements correspond to the influence of `x` over the value returned by the `hash` function. I fixed `OUTPUT_WIDTH` to have the value 15 and let `ROUNDS` vary. In order to avoid creating specific hardcoded matrices to demonstrate the values, I instead uniformly sampled the seed value from $[0, 2^{31} - 1]$ and repeated the measurement 100 times for each value of `ROUNDS`. By construction, the information flow is limited by both the number of input and output bits, i.e.,

min(ROUNDS, OUTPUT_WIDTH). In addition, the influence should be exactly that value, unless the columns of the pseudorandom matrix do not fully span the space of possible outputs.

Figure 4.1 plots my results. Exactly as predicted, the influence measurements are limited by (and usually exactly equal to) min(ROUNDS, OUTPUT_WIDTH), with smaller information flow when the matrix columns do not span all possible outputs. For example, the probability that a 15×15 binary matrix being full-rank is about 29%, and one can see that when ROUNDS = OUTPUT_WIDTH = 15, a bit less than 1/3 of the 100 pseudorandom matrices produce the full 15 bits of influence.

4.2 Comparison: TEMU

In this section, I compare my results to examples from prior work by Newsome, McCamant, and Song [25] as an evaluation of their influence measurement tool, called TEMU[4]. I show that nsqflow is able to measure information flow consistent with the findings of Newsome et al., with higher precision and better running time in about half of the cases presented. For the reader's convenience, I have included my translations into C of the relevant test programs, shown in listings 4.2 to 4.9.

```
int copy(int i) {
  int v = i;
  return v;
}
```

Listing 4.2: Copy

```
int masked_copy(int i) {
  int v = i & 0x0f;
  return v;
}
```

Listing 4.3: Masked copy

[4]Actually, it is an extended tool based on the existing TEMU, the dynamic taint analysis component of the BitBlaze binary analysis software suite [36]

```
int base = CONSTANT; // any constant
int checked_copy(int i) {
  int v;
  if (i < 16)
    v = base + i;
  else
    v = base;
  return v;
}
```

Listing 4.4: Checked copy

```
int div2(int i) {
  int v = i / 2;
  return v;
}
```

Listing 4.5: Divide by 2

```
int mul2(int i) {
  int v = i * 2;
  return v;
}
```

Listing 4.6: Multiply by 2

```
int implicit(int input) {
  int output = 0;
  if (input == 0) output = 0;
  else if (input == 1) output = 1;
  else if (input == 2) output = 2;
  /* ... */
  else if (input == 6) output = 6;
  else output = 0;
  return output;
}
```

Listing 4.7: Implicit flow

		Influence (bits)			
Name	Fig # in [25]	nsqflow	TEMU-sound	TEMU-sample	TEMU-approx-#SAT
Copy	Table 1	32	6.04–32	31.8–32	32
Masked	Table 1	4	4	-	-
Checked	Ex. 1	4	4	-	-
Div2	Table 1	31	6.58–31	30.8–31	31.7
Mul2	Table 1	31	6.58–32	30.4–31.6	31.5
Implicit	Figure 1	2.81	2.81	-	-
Popcnt	Figure 2	5.04	5.04	-	-
MixDup	Figure 2	16	6.04–32	0–28.6	15.8

Table 4.1: Information flow for programs given in [25]. The second column represents the figure number in [25] (which I have reproduced in listings 4.2 to 4.9). Ranges are given as x–y when values from [25] are approximate.

```
int popcnt(unsigned int i) {
  i = (i & 0x55555555) + ((i >> 1) & 0x55555555);
  i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
  i = (i & 0x0f0f0f0f) + ((i >> 4) & 0x0f0f0f0f);
  i = (i & 0x00ff00ff) + ((i >> 8) & 0x00ff00ff);
  int output = (i + (i >> 16)) & 0xffff;
  return output;
}
```

Listing 4.8: Population count

```
unsigned int mix_copy(unsigned int x) {
  unsigned int y = ((x >> 16) ^ x) & 0xffff;
  unsigned int output = y | (y << 16);
  return output;
}
```

Listing 4.9: Mix and duplicate

I present a comparison of my results for these programs in Table 4.1 and Table 4.2. In each case, I measured the flow of my tool from the input to the listed function's return value. The columns labeled TEMU-sound, TEMU-sample, and TEMU-approx-#SAT represent the three different complementary approaches employed by Newsome et al. TEMU-sound is the

		Running time (s)			
Name	Fig # in [25]	nsqflow	TEMU-sound	TEMU-sample	TEMU-approx-#SAT
Copy	Table 1	0.16	0.5–3.8	0.5–3.8	<30
Masked	Table 1	0.16	0.5–3.8	-	-
Checked	Ex. 1	0.16	0.5–3.8	-	-
Div2	Table 1	0.17	0.5–3.8	0.5–3.8	<30
Mul2	Table 1	0.17	0.5–3.8	0.5–3.8	<30
Implicit	Figure 1	0.17	0.5–3.8	-	-
Popcnt	Figure 2	0.17	0.5–3.8	-	-
MixDup	Figure 2	0.17	0.5–3.8	0.5–3.8	<30

Table 4.2: Running time for programs given in [25]. The second column represents the figure number in [25] (which I have reproduced in listings 4.2 to 4.9). Ranges are given as x–y when values from [25] are approximate.

first technique their tool uses, and is able to find sound lower and upper bounds on the channel capacity. When the lower and upper bounds are equal, they have found a precise result. However, when their sound technique fails to find a precise result after several seconds, their tool falls back to approximation techniques represented by the next two columns, TEMU-sample and TEMU-approx-#SAT. The column labeled TEMU-sample represents a probabilistic bound based on sampling the space of possible inputs by making queries to a constraint solver. Finally, the column labeled TEMU-approx-#SAT is their estimate using an approximate model counter to obtain a probabilistic estimate of the size of the output set. Although they do not present exact runtimes, they state that for TEMU-sound and TEMU-sampling, runtimes for the examples varied between 0.5 and 3.8 seconds, while TEMU-approx-#SAT never required more than 30 seconds.

In [25], Newsome et al. also present another set of experiments. Unfortunately, however, I was unable to reproduce these experiments using nsqflow due to the nature of their tool compared to mine. Specifically, the *RPC DCOM*, *SQL Server*, *Samba Filesystem*, *ATPhttpd*, and *synthetic switch statement* test cases measure influence over the program counter, which is beyond the scope of my tool since nsqflow operates on source code.

A limitation of my comparison with respect to running time is that, despite contacting the authors for assistance with installing and using their tool, I was unable to compile their TEMU-based tool on my own testing

machine. The figures I present are taken directly from the experiments by Newsome, McCamant, and Song [25]. As a result, the figures I present for running times should not be directly compared. One should, however, note that the testing machines had similar specifications, although mine was a slightly newer-generation machine and I also benefit from improvements made to SAT and SMT solvers since [25] was published.

4.3 Comparison: sqifc

I also compare how nsqflow performs on examples from Phan and Malacaria [26], contrasting the bounds and running times of nsqflow with those of their tool, **sqifc**[5]. The examples are CRC8, the Grade Protocol, and the Dining Cryptographers problem, which can be found in Figures 10, 12, 18 in [26], respectively. For convenience, I also provide these listings in Listings 4.10, 4.11, and 4.12.

As with the experiments from Section 4.2, I contacted the authors for assistance in installing and using their tool, sqifc. Despite making a significant effort to install it, along with the guidance of the authors, sqifc is a research prototype that is not easy to use in a general setting. As a result, I was unable to run sqifc on my own testbed. Therefore, as in Section 4.2, I present the figures reported by the authors for comparison against the values I obtained from nsqflow. Consequently, a direct running time comparison between nsqflow and sqifc cannot be made on the basis of the figures I present in this section. For the test cases to which I do compare, I note that the testing machine used by Phan and Malacaria was a newer-generation Intel i7 3.4GHz CPU with 8GB of RAM – a higher-performance machine than my own testbed.

[5] Actually, in the same paper, Phan and Malacaria also present another tool for C programs, which yielded significantly worse computational performance than sqifc in all the examples they presented. Thus, I compare only against their best tool, sqifc.

sft	0	1	2	3	4	5	6	7	8
Time (s)	0.21	0.21	0.22	0.20	0.18	0.21	0.20	0.20	0.22

Table 4.3: Running time when running nsqflow on the crc8 program from [26], for various shift values `sft`. In [26], Phan and Malacaria presented results only for *sft* values of 3 and 5 — 0.475 and 0.289 seconds, respectively. In all cases, nsqflow correctly reported $2^{8-\texttt{sft}}$ bits of influence.

```
unsigned char GetCRC8 (
        unsigned char check , unsigned char ch)
{
  int i, sft;
  for ( i = 0 ; i < 8 ; i ++ ) {
    if ( check & 0x80 ) {
      check <<=1;
      if ( ch & 0x80 ) { check = check | 0x01;}
      else { check = check & 0xfe ; }
      check = check ^ 0x85 ;
    } else {
      check <<= 1;
      if ( ch & 0x80 ) { check = check | 0x01;}
      else { check = check & 0xfe ; }
    }
    ch <<= 1;
  }
  check >>= sft;
  return check;
}
```

Listing 4.10: CRC8

The first example, *crc8*, computes an 8-bit CRC over an 8-bit value and shifts the final result by `sft` significant bits, with the source code presented in Listing 4.10. Table 4.3 presents nsqflow runtimes for `sft` values ranging from 0 to 7. In [26], results were presented only for *sft* values of 3 and 5, for which their running times using their CBMC-based sqifc tool were 0.475 seconds and 0.289 seconds, respectively. In all cases, my tool correctly reported $2^{8-\texttt{sft}}$ bits of influence. Thus I omit these figures from the table.

```
int func () {
  size_t S = 5, G = 5, i = 0, j = 0;
  size_t n = ((G -1)* S )+1 , sum = 0;
  size_t numbers [S], announcements [S], h[S];
  for (i=0; i<S; i++) h[i] = nondet_int()%G;
  for (i = 0; i < S; i++)
    numbers [i] = nondet_int() % n;
  while (i <S) {
    j=0;
    while (j <G) {
      if (h[i ]== j)
      announcements [i] =
        (j+ numbers [i]- numbers [(i+1)%S]) % n;
      j=j+1;
    }
    i=i+1;
  }
  for (i = 0; i < S; i ++)
    sum += announcements[i];
  return sum % n;
}
```

Listing 4.11: Grade protocol

The second example to which I compare is the grade protocol program given in Section 3.5 of [26] and copied in Listing 4.11. In [26], Phan and Malacaria vary the number of *students* in the protocol, and the number of possible *grades*, between 2 and 5, presenting sqifc information flow measurements and running times for these values. I ran nsqflow for the same values of *students* and *grades* as Phan and Malacaria, and present the information flow measurements obtained from nsqflow in Table 4.4 and nsqflow's running times, along with sqifc's running times from [26] in Table 4.5. In addition, I ran nsqflow for 16 and 24 *students*, and for 16 and 24 possible *grades*. Table 4.4 and Table 4.5 present nsqflow's information flow measurement and running times for these values. The running time measurements demonstrate the advantage of nsqflow; my tool runs in 10.43 seconds for values of *students*=24 and *grades*=24, whereas the CBMC-based sqifc takes 40.4 seconds for values of *students*=5 and *grades*=5. In all the cases in which both tools ran to completion, the number of bits reported by nsqflow and sqifc were equal.

	2	**3**	**4**	**5**	**16**	**24**
2	1.585	2.0	2.322	2.585	4.087	4.644
3	2.322	2.807	3.17	3.459	5.044	5.615
4	2.807	3.322	3.7	4.0	5.615	6.19
5	3.17	3.7	4.087	4.392	6.022	6.6
16	4.954	5.524	5.931	6.248	7.913	8.496
24	5.555	6.129	6.539	6.858	8.527	9.111

Table 4.4: Grade protocol information flow results. The row index represents the number of students, and the column index represents the number of distinct grade values each student can take. **nsqflow** and **sqifc** reported the same figures for all cases in which both tools ran to completion. The two bottom rows, and the two rightmost columns, represent results computed by **nsqflow** for which no equivalent results from **sqifc** were available.

Grades	**Students**											
	2		**3**		**4**		**5**		**16**		**24**	
	nsq	**sqif**	**nsq**	**sqif**	**nsq**	**sqif**	**nsq**	**sqif**	**nsq**	**sqif**	**nsq**	**sqif**
2	0.36	5.66	0.45	7.03	0.61	10.77	0.86	9.47	1.22	-	2.83	-
3	0.34	9.15	0.49	11.60	0.70	17.99	1.17	20.93	2.03	-	3.12	-
4	0.49	10.10	0.60	16.87	0.80	21.87	1.15	18.67	2.43	-	3.10	-
5	0.85	14.64	0.57	20.67	0.82	33.30	1.63	40.40	2.55	-	3.67	-
16	1.24	-	1.53	-	1.79	-	3.12	-	4.14	-	5.42	-
24	1.59	-	1.86	-	1.97	-	3.71	-	5.11	-	9.88	-

Table 4.5: Grade protocol running time results. The columns labeled **nsq** present **nsqflow**'s running times, and the columns labeled **sqifc** present **sqifc**'s running times from [26]. All running times are measured in seconds. The row index represent the number of students, and the column index represents the number of distinct grade values that each student can take.

The third example for comparison is the well-known dining cryptographers protocol, shown in Listing 4.12 for convenience. I ran nsqflow on the same C code as given by Phan and Malacaria [26] with the same parameter values (various values up to 300) for n, the number of cryptographers, and present the results in Table 4.6. I obtained the same influence measurements presented by Phan and Malacaria [26]. I found the running time of nsqflow to be much lower, with nsqflow requiring only 3.12 seconds for $n = 300$, compared to the reported 3326.9 seconds for sqifc. I also ran my tool for larger values of n, specifically $n = 1000$ and $n = 2000$. For $n = 2000$, nsqflow still required only 167 seconds.

```
size_t func () {
  size_t N = 5, output = 0, i = 0;
  size_t coin[N], obscoin[2], decl[N];
  size_t h;
  h = nondet_uchar () % (N+1);
  for (i = 0; i < N; i++) {
    coin [i] = nondet_uchar() % 2;
  }
  for (i = 0; i < N; i++) {
    decl[i] = coin[i] ^ coin [(i+1)%N];
    if (h == i+1) {
      decl[i] = !decl[i];
    }
    i = i+1;
  }
  for (i = 0; i < N; i++) {
    output = output + decl[c];
  }
  return output;
}
```

Listing 4.12: Dining cryptographers

Phan and Malacaria also presented results for three other experiments, all of which I was unfortunately unable to reproduce. Specifically, I was unable to replicate the *Tax Record* test case as it is implemented in the Java-like verification language ABS [10], and I was unable to translate it into equivalent C. Also from Phan and Malacaria's work [26], I was unable to replicate the test cases *CVE-2011-2208* and *CVE-2011-1078*, despite being C code from the Linux kernel. These test cases were modified by the authors,

n	3	4	5	6	50	100	200	300	1000	2000
Bits	2.0	2.32	2.58*	2.81	5.67	6.66	7.65	8.23	9.97	10.97
nsq	0.25	0.27	0.33	0.32	0.75	1.41	1.85	3.01	16.24	166.75
sqif	2.15	3.50	3.63	18.63	46.97	158.52	587.67	3326.92	–	–

Table 4.6: Dining cryptographers protocol results. The row labeled **Bits** represents the number of bits measured by **nsqflow**. The row labeled **nsq** represents **nsqflow**'s running time, and the row labeled **sqif** represents **sqifc**'s running time. **n** is the number of cryptographers, and running time is measured in seconds. The starred value 2.58 differs from Phan and Malacaria's reported 2.59. I attribute this to roundoff error during the feasible set size to information flow calculation.

who employed manual program slicing and system call modeling to reduce the size of the program. I contacted the authors, but they did not have ready access to the source code at the time, so I was unable to include these examples in my experiments.

Because the tools underlying nsqflow make use of randomization in some of their decisions and heuristics, running times are potentially sensitive to these choices. Thus, I also measured how tightly clustered the running times are. I ran all the examples from Section 4.2 and Section 4.3 from 20 to 100 times and measured the interdecile range for the running times. I found that the running times are tightly clustered, with ranges of 0.01-0.04 seconds for the tiny examples sensitive to background CPU tasks, and on the order of 1-5% for the larger experiments of this set. Full results may be found in Table 4.7.

4.4 Larger Experiments

In this section, I present further experiments on real C code. The experiments range from 1915 to 23759 lines of code, with the output size ranging from 32 to 32768 bytes, and the measured influence ranging from 0 to 32752 bits, largely depending on the output chosen. Running times ranged from 2 minutes for the 2600-line darkHttpd, to 3 and a half hours for the 20000-line xinetd. Lines of code are measured as reported by the UNIX `wc -l` tool on `.c` and `.h` files in the source directories and subdirectories. While this does not measure the number of included standard library headers or account for comments, it is the simplest way to get a general idea about the size of the program. The reader will notice the large proportion of http and ftp servers and clients. This choice was somewhat accidental, and resulted from my desire to use open-source C programs (found on SourceForge or similar) that

Test	Number of Runs	Median Running Time (s)	Interdecile Range (s)
Copy	100	0.16	0.01
Masked copy	100	0.16	0.01
Checked copy	100	0.16	0.02
Divide by 2	100	0.17	0.02
Multiply by 2	100	0.17	0.02
Implicit flow	100	0.17	0.02
Population count	100	0.17	0.01
Mix and duplicate	100	0.17	0.01
CRC8, sft=0	100	0.21	0.03
CRC8, sft=1	100	0.21	0.02
CRC8, sft=2	100	0.22	0.04
CRC8, sft=3	100	0.20	0.03
CRC8, sft=4	100	0.18	0.02
CRC8, sft=5	100	0.21	0.02
CRC8, sft=6	100	0.20	0.02
CRC8, sft=7	100	0.20	0.02
CRC8, sft=8	100	0.22	0.02
DinCryptos, n=3	100	0.25	0.02
DinCryptos, n=4	100	0.27	0.02
DinCryptos, n=5	100	0.33	0.03
DinCryptos, n=6	100	0.32	0.03
DinCryptos, n=50	100	0.75	0.04
DinCryptos, n=100	100	1.41	0.05
DinCryptos, n=200	100	1.85	0.08
DinCryptos, n=300	100	3.01	0.07
DinCryptos, n=1000	50	16.24	0.60
DinCryptos, n=2000	20	166.75	2.24

Table 4.7: Interdecile range (in seconds) for the comparison programs. The range is reported as the difference between the 90^{th} and 10^{th} percentiles.

were of the right scope for nsqflow (roughly 10K LOC or under), and ideally ones that were web-facing or otherwise interesting from a typical security point-of-view. My tool, however, is not limited to programs of this nature. For the full technical details of the experiments, along with results, please consult Table 4.8. A brief description of each program I tested is found in the list below.

muHttpd Forking HTTP server with basic sanitizer function. The exit point/output variable is a buffer to which data from the file system is copied before it is sent back over the web in order to service the request. **File**: `request.c`; **Function**: `handle_request`; **Variable**: `buf`.

awHttpd Forking HTTP server. The exit point is a buffer to write back following an http request. **File**: `proc.c`; **Function**: `procsendfile`; **Variable**: `cn->databuf`.

darkHttpd HTTP server. The exit point is a server name (string) pointer set from the command-line at program start-up, and should not be able to be modified by its (network) inputs. **File**: `darkhttpd.c`; **Function**: `main`; **Variable**: `wwwroot[0]`.

xinetd The Linux Extended Internet Services Daemon, version 2.3.15. The exit point is a newly-allocated server object based on an old (input) one. This test checks whether function side effects can block any possible values for the output variable (the variable to which the contents input variables' addresses are copied). **File**: `server.c`; **Function**: `server_alloc`; **Variable**: `<return value>`.

FTP Client Command-line FTP client. The exit point is a character buffer containing a 3-character FTP return code to be interpreted plus a null character. This experiment tests whether all potential values are copied to the output buffer and that there is nothing written past the end of the array that would replace the null character (e.g. off-by-1 error or buffer overflow). **File**: `ClientFTP.c`; **Function**: `interpreteReponse`; **Variable**: `rep`.

thttpd Throttling HTTP server, version 2.26. The exit point is a parameter used for throttling connection speed. This experiment verifies whether the state of the connection, which may be influenced by an attacker, can affect the throttling parameter to a large degree, or only

restricts it to a small set of possible values. **File**: `thttpd.c`; **Function**: `handle_send`; **Variable**: `max_bytes`.

Thy Thy HTTP server, version 0.9.4. The output variable is an internal code representing the result of a parse, of which a small set of values should be possible. This tests whether an attacker can force this return code to take on an arbitrary value, rather than the narrowing of the information flow intended by the code. **File**: `http.c`; **Function**: `http_request_parse`; **Variable**: `<return value>`.

ftpRelay2 FTPRelay ftp library, version 2. The output variable is a buffer into which network data is read. This experiment tests whether or not its simple input sanitizer check constrains the data received. **File**: `childproc.c`; **Function**: `ClientToServer`; **Variable**: `RecvBuffer`.

httpserver1 Small http server, version 0.3. The output variable is an internal code depending on data read from the network, as part of a struct. This experiment aims to test for the presence or absence of a simple buffer overflow — whether or not adjacent data in the struct is clobbered when writing it. **File**: `httpsmain.c`; **Function**: `set_incoming`; **Variable**: `sv.rc`.

ftpslib1 FTP library, version 0.2. This experiment aims to test for the presence or absence of a simple buffer overflow - whether or not adjacent data in the struct is clobbered when writing it (similarly-structured program to the above httpserver1). **File**: `ftplib.c`; **Function**: `set_mode`; **Variable**: `fcode`.

Bugs Found: For two of the experiments, httpserver1 and ftpslib1, the large influence reported turned out to be actual security vulnerabilities in the software. In both these cases, a buffer overflow was present when writing the value adjacent to the measured variable. Both these bugs were previously undisclosed, but confirmed by the developers.

4.5 CoreUtils

I also selected a subset of the 30 smallest programs, as measured by lines of code, from the GNU CoreUtils suite, version 8.23. I chose the flow being from `argv[1]` to the return code of the main function in all cases, bounding the size of the argv string to 256 bytes. I selected these examples arbitrarily to demonstrate nsqflow's robustness for use as a completely automated tool

Test	LOC	Sliced LOC	Sliced LOC Reduction	Exit Point Size (bits)	Running time	Symbolic execution time (% of total)	Model counting time (% of total)	RAM Usage (MB)	Influence (bits)
mu-Httpd	2268	1996	12%	32768	7m	84%	16%	258	32752
aw-Httpd	2257	1602	29%	8192	22m	74%	26%	347	8192
dark-Httpd	2599	837	68%	32	1m 52s	85%	15%	494	0
xinetd	23759	6320	73%	256	3h 39m	90%	10%	1818	256
FTP-Client	1915	440	77%	32	2h 54m	81%	19%	449	24
thttpd	11155	3681	67%	32	2h 1m	73%	27%	1330	1
Thy	17134	4968	71%	32	3h 13m	82%	18%	1653	1
ftp-Relay2	1080	944	13%	8192	timeout	99%	1%	≥1745	≥3073
http-server1	1433	978	32%	32	2h 16m	83%	17%	319	32
ftp-slib1	1611	992	38%	32	1h 28m	84%	16%	336	32

Table 4.8: Larger experiment results for the programs presented in Section 4.4. LOC stands for lines of code. The **ftpRelay2** entry is included as an example of my tool's limitations — although this program is not fundamentally different from others in the list, **nsqflow** is unable to complete its analysis despite its relatively small size, timing out after 20 hours. Program slicing effectiveness is measured in LOC before preprocessing. For RAM usage, the reader should note that **nsqflow** has a maximum RAM usage of 2000MB that is inherited from Kite.

rather than one requiring careful and thoughtful developer annotations. I ran nsqflow on these program once each, with a timeout of 60 minutes for each program. 20 of the 30 completed in 60 seconds or less, with only 2 of 30 timing out after 60 minutes. Influence measurements ranged from 0 to 2 bits for the instances running to completion. Full details of these results may be found in Table 4.9.

In addition, I used a uniform random number generator to select a subset of 30 programs from CoreUtils, and ran nsqflow on them in the same fashion. This resulted in 9 programs already included in the 30 smallest, leaving us with 21 new CoreUtils[6]. 9 of the 21 tests completed in 60 seconds or less, with only 4 of the 21 timing out after 60 minutes. Influence measurements were 1 or 1.585 bits for the instances running to completion. For CoreUtils, I omit RAM figures, as I have found that memory usage has not been a problem. I present the full details of these results in Table 4.10, with the ones appearing in the 30-smallest benchmark removed, leaving a total of 21 programs).

4.6 Performance Discussion

In this section, I discuss issues related to performance and optimizations, recognizing that a reader may wonder which optimizations and components of nsqflow's toolchain are necessary for nsqflow to scale well.

Empirically, I have found that my pointer analysis has been crucial to scale to the size of tests I have presented. I ran the instances described in Section 4.4 with nsqflow's pointer analysis disabled, and a timeout of 8 hours. None of these instances managed to run to completion, with all but muHttpd running into the intrinsic 2000 MB RAM usage limit that nsqflow has inherited from Kite.

Additionally, the underlying symbolic execution engine's ability to learn from infeasible path constraints is one of its defining features. Thus, an interesting comparison is between nsqflow using Kite with its path learning enabled, and nsqflow using Kite with path learning disabled. As described earlier, the pointer analysis I have implemented interacts with the path learning. As I have just shown above, pointer-heavy code tends to make nsqflow unable to complete. As a consequence, I am unable to meaningfully present figures for nsqflow with both path learning and pointer analysis turned off.

[6] I chose to stick with the 21 programs left over, so as to maintain an unbiased random selection.

Test	Lines of Code	Influence (bits)	Time (seconds)
basename	190	1	21
dirname	137	1	8
echo	273	0	10
env	164	≥1.585	timed out
groups	141	1	15
hostid	89	1	11
hostname	117	1	4
link	95	1	4
logname	87	1	8
mkdir	307	1	30
mkfifo	182	1	622
mknod	274	1	33
nice	223	2	70
nohup	241	≥1.585	timed out
nproc	134	1	17
printenv	155	1.585	23
readlink	179	1	65
realpath	278	1	88
rmdir	253	1	546
runcon	267	2	21
sleep	150	1	71
sum	275	1	56
sync	74	1.585	13
tee	221	1	85
true	81	0	3
tty	125	2	4
unlink	90	1	4
users	152	1	61
whoami	95	1	16
yes	89	1	5

Table 4.9: Information flow for the set of the 30 smallest CoreUtils. Information flow is measured to the return value of the `main` function. Each program is run with a timeout of 60 minutes.

Test	Lines of Code	Influence (bits)	Time (seconds)
base64	326	1	39
cat	785	1	214
chmod	571	1	105
chown	331	1	59
comm	450	1	28
cp	1221	1	329
expand	431	1	102
factor	2548	$\geq$1	timed out
fmt	1042	1	756
install	1047	$\geq$1	timed out
md5sum	879	$\geq$1	timed out
nl	617	1	36
numfmt	1523	1.585	81
pathchk	424	1	38
shuf	627	1	238
sort	4751	$\geq$0	timed out
stat	1580	1	39
touch	438	1	26
truncate	425	1	41
uname	376	1	24
wc	802	1	153

Table 4.10: Information flow for the set of randomly-selected CoreUtils. Information flow is measured to the return value of the `main` function. Each program is run with a timeout of 60 minutes.

Moreover, the reader may have noticed that not every source line of a program may be relevant to measuring a specific flow. More generally, nsqflow is able to effectively avoid analyzing large parts of the input program through the technique of program slicing [41]. I leverage the program slicing implemented in Kite, and also a compound slicer using off-the-shelf Frama-C [5] and LLVMSlicer [34]. I note that the program slicing performed is entirely automatic, which is in contrast to results presented in, for instance, Phan and Malacaria's work [26], which involves manual slicing. For the set of benchmarks I have presented, I have found slicing to be very effective. I measured the degree of code eliminated through program slicing as a way to demonstrate that, although the nsqflow toolchain must solve computationally difficult problems, existing heuristics can often reduce the size of the problems in practice, thus making them tractable. Indeed, nsqflow's ability to complete is sensitive to the effectiveness of the slicing on a given program. Because the LLVM-based slicing does not produce valid source, I approximate the number of source lines sliced by measuring the fraction of LLVM bitcode instructions sliced out. I found that these figures ranged between 12% to 73% for the programs used in the experiments presented in Section 4.4. Full details were presented in Table 4.8.

Furthermore, a reasonable question to pose might be where the bottleneck is typically found in a run of nsqflow. I have previously hinted at the surprising result that model counting has not been the most expensive step in nsqflow's toolchain, and I present data to support this claim. Although my initial expectations were that nsqflow would nearly always be limited by the model counting, I now believe that my particular variant of the more general #SAT problem is responsible for the relatively short time spent inside the model counter. I present a breakdown of the total running time spent in each component of nsqflow for the experiments from Section 4.4. I found that for these, the proportion of time spent performing model counting accounts for 10% to 27% of total running time. Table 4.8 lists this information as a percentage of total running time, to two significant figures.

Chapter 5

Related Work

There is a large body of literature about quantitative information flow, much of it more theoretical than the scope of this thesis [2, 16, 17, 31, 35, 39, 42]. In this chapter, I survey work that, like nsqflow, aims to provide a practical tool to compute QIF. I also survey techniques from software and hardware verification that are useful in QIF.

5.1 Network-Flow-Based QIF

In [21], McCamant and Ernst present a scalable approach to measuring QIF using a network flows approach. Unlike the problem I consider in this thesis, measuring the degree of influence of a program's inputs over its outputs, they address a related confidentiality problem, quantifying the information an attacker can learn about a program's inputs by observing its outputs. They model information channels in a program as a network of finite-capacity pipes and reduce the measurement problem to a network flow computation, solving it using standard network flow algorithms.

Specifically, they construct a graph in which the edges represent variables (including inputs) in the program, with weights corresponding to the bit-widths of the variables, and the nodes represent program instructions on these variables, with the in-degrees of the nodes representing the number of arguments needed by the corresponding instructions. The graph is acyclic, and edges always point from older (in program instruction order) nodes to newer ones. The set of all secret inputs and the set of all public outputs are represented by the source and sink nodes, respectively.

In order to handle implicit flows, as in the case of a multi-way branch, McCamant and Ernst use static analysis to infer *enclosure regions*, which declare locations that the code enclosed in enclosure regions may write to. These enclosures become special aggregate nodes in the graph, with weights on edges in and out of these nodes corresponding to the number of different executions (implicit flows) through the aggregate nodes, equal to the base-2 logarithm of the number of branches within enclosures.

To obtain the edge weights, the authors use dynamic tainting at the level of bits, maintaining which bits are certainly leaked, which are certainly not leaked, and which bits are unknown. In order to maintain soundness, unknown bits are treated conservatively and assumed to be leaked. Consequently, the result is typically an overapproximation, which tends to be cruder as the program size increases. Finally, using standard network flow algorithms, McCamant and Ernst's tool arrives at a final QIF measurement. Their graph construction is efficient, as are the network flow algorithms they use. As a result, their tool is able to effectively scale to programs consisting of 500K or more lines of code, at the cost of precision — indeed, their approximation is crude when compared to the QIF reported by nsqflow, as their approach can only compute an upper bound on the channel capacity.

To the best of my knowledge, the work by McCamant and Ernst has been the only approach to date, that attempts to measure QIF through programs based on network flows. While not as precise as nsqflow, the scalability of their approach is impressive and I believe that further work in this area is a promising direction for future research in measuring QIF through programs.

5.2 Symbolic-Execution-Based QIF

In [25], Newsome, McCamant, and Song present a way to measure channel capacity, addressing the same fundamental problem as I do in this thesis. In a manner somewhat reminiscent of nsqflow's, they use a symbolic execution engine to obtain a formula over the program inputs and outputs that represents the entire program, expressed in first order logic over bitvectors and arrays. Once this formula is obtained, their tool makes repeated queries to a constraint solver in order to characterize the set of possible output values the program can take. Using a heuristic approach, they make queries about whether certain values are feasible or found within certain value ranges, until the number of feasible values they have discovered crosses a desired threshold. Theoretically, this step is precise if run to completion, but is very computationally expensive, and the tool is typically only able to check a small number of values (typically 64 or 128) before stopping and reporting a number of bits (6 or 7) as a lower bound. In order to combat the large computational cost associated with this step, the authors also implement an approach based on approximate model counting in order to obtain an approximation of the number of solutions. However, as I have shown in Section 4.2, their precision is quite low even for small examples, whereas nsqflow reports a precise and exact influence measurement. Unlike nsqflow, which

exhaustively explores every program path, their approach only operates on a single path. Therefore, the feasible value set they obtain is only a subset of the whole program's, and their resulting influence measurement is only a *lower* bound on the channel capacity.

Newsome, McCamant, and Song also adopt a different, but even less precise, method to scale to much larger program sizes. Specifically, the authors consider a return-oriented attack [30] by tracking the inputs' influence over the program counter. Using this program counter technique along with conservative lower bounds and approximating model counting, their tool is able to obtain approximate information flow bounds that scales to very large programs, on the order of a million lines of code. Their technique has the added benefit of operating on binaries, thus eliminating the need for the program's source as nsqflow requires. However, the precision of their approach is very low for such large programs.

Most similar to my approach is the work by Phan and Malacaria described in [26, 27]. As in this thesis, Phan and Malacaria also use an approach based on symbolic execution and constraint solving to solve the problem of obtaining the channel capacity between a program's inputs and its outputs. Their tool, sqifc, symbolically executes a program, maintaining a running total of the number of solutions as it executes. This running total is updated with multiple feasible values when symbolic execution determines a path to be feasible, and sqifc counts the number of new solutions discovered as a result, for each possible program path.

In this way, Phan and Malacaria integrate model counting with symbolic execution — the first of two fundamental differences between my work and theirs. In contrast, my approach uses symbolic execution to produce a CNF representation of the program, and then subsequently applies an efficient (subset) model counter to that CNF. To facilitate their integrated model counting, Phan and Malacaria introduce #DPLL(T) model counters. A #DPLL(T) solver counts the number of distinct satisfying solutions to the Boolean skeleton of an SMT formula.[7] The SMT-based framework they present is very flexible, allowing them to represent existing model checking and symbolic execuction engines as theory solvers. However, as the authors write, #DPLL(T) model counters must explicitly enumerate each satisfying solution to the Boolean skeleton individually, as each satisfying assignment to the Boolean skeleton must also be checked for satisfiability against the

[7]Note that whereas an SMT formula may in some cases have an unbounded number of distinct models (for example, if the formula includes unconstrained real numbers), there are at most a finite number of distinct satisfying assignments to the Boolean skeleton of any SMT formula.

theory solvers. In contrast, the #SubSAT solver in nsqflow is able to generalize individual satisfying assignments, often allowing the model counting step to complete quickly even for large instances and large, multi-byte output values. Indeed, as I have shown in Chapter 4,in most of my experiments, symbolic execution was the dominant cost, while the model counting step completed relatively quickly.

Second, Phan and Malacaria's work differs from mine in the symbolic execution step. Like nsqflow, if their symbolic execution runs to completion, their result will be a sound and complete bound on the total number of solutions, and therefore be a precise measurement of the channel capacity. However, their tool does not leverage learning CFG information during symbolic execution, and therefore cannot prune paths as effectively as nsqflow can.

As a consequence of these key differences in symbolic execution and model counting, their results, while precise, scale to programs that span only a few hundred lines of C following manual modeling and program slicing. As I have shown in Section 4.3, nsqflow is also precise but scales to much larger programs, and has far lower running times for small programs.

To the best of my knowledge, there has not been other published work in symbolic-execution-based QIF apart from the work I have cited and described here.

Chapter 6

Conclusion

6.1 Contributions

I have described an approach to measuring the QIF, specifically the channel capacity, through software. My work is built upon state-of-the-art symbolic execution techniques and model counting. While previous approaches either scaled to very large programs but provided coarse approximations to QIF [21, 25], or were precise but only scaled to hundreds of lines [2, 22, 25, 26], I demonstrate that it is possible to be highly precise and scale to tens of thousands of lines of real C code.

My implementation of this approach, nsqflow, is fully automatic and tracks both explicit and implicit information flows through all (finitely bounded) program paths, with high precision. I show how pointer analysis can be added to existing symbolic execution engines in order to make feasible the analysis of code with pointers. I describe a new variant of the exact model counting problem, which I have called #SubSAT, that is tractable for practical examples compared to the full model counting problem, and makes feasible model counting specifically for QIF.

6.2 Future Work

There are many directions for future work. A near term possibility would be to expand the applications of nsqflow. For example, I could extend the tool to handle C++, which should be straightforward given that LLVM can handle C++ already. Or, capitalizing on the completely automatic nature of the tool, one could imagine background application of nsqflow on a massive scale, similar to fuzz testing or regression testing, to flag suspicious information flows. One way to implement such an application could be to use an automatic selection process, perhaps based on static analysis and heuristics, to automatically select variables over which to measure the influence. The application could be continuously running nsqflow to measure influence over these automatically-selected output variables, stopping when

a minimum threshold policy (similar to the one described in Section 3.1.3) for the output variables is exceeded (perhaps computed automatically as a fraction of the output variable's bit width).

In the opposite direction, it is possible to extend nsqflow to compute more expensive, but even more valuable information. For example, it might be possible to explicitly formalize side-channel information [17], and then the symbolic execution engine would know that calls to functions like `time()` must behave in certain ways, thereby capturing side-channel information via nsqflow's existing accounting of implicit information flows. For a very different example, I could extend the model counting to enumerate the input equivalence classes for each possible output value, rather than just the values. This is a much more expensive computation, but allows precisely computing alternative information flow metrics based on Shannon entropy, min-entropy, and guessing entropy, for non-uniform distributions [2], rather than only channel capacity as nsqflow currently computes.

An interesting theoretical question is how to deal with the distinction between attacker-controlled and auxillary inputs. Adopting the notation from Newsome, McCamant, and Song [25], consider a computation $P(I, I_{aux}) \rightarrow V$, where V is the output set, I represents the attacker-controlled input, and I_{aux} is the auxiliary input. A reasonable extension to the channel capacity model would be that the attacker cannot control the entire set of inputs. Rather, the attacker can only *observe* the value of I_{aux}. With this formulation, one can frame both channel capacity and this alternative definition in the following manner, respectively:

$$\operatorname*{Count}_{V}(\exists I, I_{aux} \mid P(I, I_{aux}) = V)$$

$$\max_{I_{aux}} (\operatorname*{Count}_{V}(\exists I \mid P(I, I_{aux}) = V))$$

The latter is fundamentally a more precise model than the former, and is appropriate as it represents the worst-case internal program state to give the attacker the greatest degree of control over the output variable. Specifically, the former is problematic as it may overcount the number of output values to which the adversary can purposefully steer the computation. In contrast, in Newsome et al. [25], I_{aux} is set to an arbitrary fixed value, which may *undercount* the adversary's influence — some other fixed value for I_{aux} may give the adversary more influence. This motivates the latter definition: It represents the worst-case internal program state to give the attacker the greatest degree of control over the output variable. As such, this is a natural definition of influence in a secu-

rity context when there are auxiliary inputs not under the attacker's control.

In another direction, nsqflow would benefit greatly from a more robust pointer analysis. Indeed, in Section 4.4 and Section 4.5, nsqflow was unable to run to completion on some of the programs largely due to the shortcomings of nsqflow's current pointer analysis. Specifically, DSA did not return sufficient information to allow Kite to prune large parts of the search space that resulted from the heavy use of pointers in the programs. As the current analysis, based on LLVM's existing DSA, is simplistic, a promising future direction is to more efficiently utilize points-to set information, such as the approach presented in [28].

Finally, while the model counting has not generally been a problem for the experiments I have presented, model counting is a computationally hard problem from a theoretical perspective. There is recent work on efficient, approximate model counting, with provable bounds. [6] In some preliminary experiments, I have not yet found approximate model counting to be beneficial, but these are recent results and additional progress in this area is likely.

Bibliography

[1] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, volume 9, pages 399–404, 2009.

[2] Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 141–153. IEEE, 2009.

[3] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[4] Thierry Castell. Computation of prime implicates and prime implicants by a variant of the Davis and Putnam procedure. In *Tools with Artificial Intelligence, 1996., Proceedings Eighth IEEE International Conference on*, pages 428–429. IEEE, 1996.

[5] CEA-LIST and INRIA-Futurs. Frama C Slicer. `http://frama-c.com/slicing.html`, 2014. Accessed October 14, 2015.

[6] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. A scalable approximate model counter. In *Principles and Practice of Constraint Programming*, pages 200–216. Springer, 2013.

[7] David R Chase, Mark Wegman, and F Kenneth Zadeck. *Analysis of Pointers and Structures*, volume 25. ACM, 1990.

[8] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 46–52. IEEE, 2013.

[9] Niklas Eén and Niklas Sörensson. MiniSat: A SAT solver with conflict-clause minimization. *SAT*, 5, 2005.

[10] Reiner Hähnle, Einar B Johnsen, Bjarte M Østvold, Jan Schäfer, Martin Steffen, and Arild B Torjusen. Deliverable D1.1A: Report on the Core

ABS language and methodology, part A. Technical Report FP7-231620, HATS Project, 2010.

[11] William G Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[12] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[13] Arthur Hicken. SQL injection hall of shame. `http://codecurmudgeon.com/wp/sql-injection-hall-of-shame/`, 2015. Accessed October 14, 2015.

[14] NTT Innovation Institute. NTT Group global threat intelligence report, 2014.

[15] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[16] Vladimir Klebanov. Precise quantitative information flow analysis using symbolic model counting. In *Proceedings of the International Workshop on Quantitative Aspects in Security Assurance*. Citeseer, 2012.

[17] Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 286–296, 2007.

[18] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[19] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM SIGPLAN Notices*, volume 42, pages 278–289. ACM, 2007.

[20] Vasco M Manquinho, Paulo F Flores, João P Marques Silva, and Arlindo L Oliveira. Prime implicant computation using satisfiability algo-

rithms. In *Tools with Artificial Intelligence, 1997. Proceedings., Ninth IEEE International Conference on*, pages 232–239. IEEE, 1997.

[21] Stephen McCamant and Michael D Ernst. Quantitative information flow as network flow capacity. *ACM SIGPLAN Notices*, 43(6):193–205, 2008.

[22] Ziyuan Meng and Geoffrey Smith. Calculating bounds on information leakage using two-bit patterns. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, page 1. ACM, 2011.

[23] Alan Mishchenko and Robert K Brayton. SAT-based complete don't-care computation for network optimization. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 412–417. IEEE, 2005.

[24] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[25] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 73–85. ACM, 2009.

[26] Quoc-Sang Phan and Pasquale Malacaria. Abstract model counting: A novel approach for quantification of information leaks. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, pages 283–292. ACM, 2014.

[27] Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S Păsăreanu. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[28] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15). USENIX Association*, 2015.

[29] Darsh P Ranjan, Daijue Tang, and Sharad Malik. A comparative study of 2QBF algorithms. In *SAT*, pages 292–305, 2004.

[30] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[31] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[32] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. *SAT*, 4:7th, 2004.

[33] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.

[34] Jiri Slaby. LLVM static slicer. `https://github.com/jirislaby/LLVMSlicer`, 2014. Accessed October 14, 2015.

[35] Geoffrey Smith. On the foundations of quantitative information flow. In *Foundations of Software Science and Computational Structures (FOSSAC)*, pages 288–302, 2009.

[36] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Information Systems Security*, pages 1–25. Springer, 2008.

[37] Celina G Val. Conflict-driven symbolic execution : How to learn to get better. Master's thesis, The University of British Columbia, March 2014.

[38] Celina G Val, Michael A Enescu, Sam Bayless, William Aiello, and Alan J Hu. Precisely measuring quantitative information flow: 10K lines of code and beyond. In *European Symposium on Security and Privacy, 2015*. IEEE, 2015.

[39] Pavol Černý, Krishnendu Chatterjee, and Thomas A Henzinger. The complexity of quantitative information flow problems. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 205–217. IEEE, 2011.

[40] Larry Wall and Schwartz Randal L. *Programming Perl*. O'Reilly and Associates, 1991.

[41] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

[42] H. Yasuoka and T. Terauchi. Quantitative information flow — verification hardness and possibilities. In *Computer Security Foundations Symposium (CSF), IEEE 23rd*, pages 15–27, 2010.