Scalable Approximation of Quantitative Information Flow in Programs

Fabrizio Biondi[1], Michael A. Enescu[2], Annelie Heuser[3], Axel Legay[2], Kuldeep S. Meel[4], Jean Quilbeuf[2]

[1] CentraleSupélec, France
fabrizio.biondi@inria.fr
[2] Inria, France
{mihai.enescu,axel.legay,jean.quilbeuf}@inria.fr
[3] CNRS/IRISA, France
annelie.heuser@irisa.fr
[4] National University of Singapore, Singapore
meel@comp.nus.edu.sg

Abstract. Quantitative information flow measurement techniques have been proven to be successful in detecting leakage of confidential information from programs. Modern approaches are based on formal methods, relying on program analysis to produce a SAT formula representing the program's behavior, and model counting to measure the possible information flow. However, while program analysis scales to large codebases like the OpenSSL project, the formulas produced are too complex for analysis with *precise* model counting. In this paper we use the *approximate* model counter ApproxMC2 to quantify information flow. We show that ApproxMC2 is able to provide a large performance increase for a very small loss of precision, allowing the analysis of SAT formulas produced from complex code. We call the resulting technique ApproxFlow and test it on a large set of benchmarks against the state of the art. Finally, we show that ApproxFlow can evaluate the leakage incurred by the Heartbleed OpenSSL bug, contrarily to the state of the art.

1 Introduction

Finding vulnerabilities in programs is fundamental for producing robust programs as well as for guaranteeing user security and data confidentiality. Due to the increasing complexity of software systems, automated techniques must be deployed to assist architects and engineers in verifying the quality of their code. Among these, *quantitative* techniques have been shown to effectively aid in detecting complex vulnerabilities.

Quantitative information flow (QIF) computation [13] is a powerful quantitative technique to detect information leakage directly at code level. QIF leverages information theory to measure the flow of information between different variables of the program. An unexpectedly large flow of information may characterize a potential leakage of information. In practice, this technique relies on the following:

the maximum amount of information that can leak from a function (known as *channel capacity*) is the logarithm of the number of distinct outputs that the function can produce [16].

Recently, QIF computation based on program analysis and model counting has effectively analyzed codebases of tens of thousands of lines of C code [31]. This technique proceeds as follows. A specific fragment of the program (e.g. a function, or the whole program) is modeled as an information-theoretic channel from its input to its output. Program analysis techniques such as symbolic execution or model checking are used to explore the possible executions of the fragment. Program analysis produces a set of constraints that characterizes these executions. Afterwards, a *model counter* is used to determine the number of distinct outputs of the fragment (e.g. the return values of the function, or the outputs of the program). Finally, the base-2 logarithm of the number of possible outputs gives us the channel capacity in bits.

However, even small programs can result in sets of constraints that are difficult to model count. Complex constraints can result, for instance, from complex program constructions such as pointers in C code. As a result, QIF computation is still not able to discover real-world, high-value security vulnerabilities.

In particular, we consider the analysis of the OpenSSL Heartbleed vulnerability [1] to be an achievable target for QIF computation, and aim to analyze vulnerabilities of this complexity. Channel capacity can be used to detect information leakage in cases like Heartbleed. For instance, if the input of a function has a capacity of 6 bits and the output a capacity of 8 bits, then the function has unexpected behavior. Further investigation can determine the origin of the information that is unaccounted for, e.g. restricted memory that the function is not supposed to have access to. The technique has been shown to be able to help detect and confirm bugs in software [26, 24, 27], and to signal to a developer that there may be bugs in a particular part of the software. Indeed, QIF-based techniques, while not foolproof, can use the a large information flow to a particular part of the program as a hint to a developer in order to narrow down where to look for bugs [31].

However, the model counting step of the procedure is very computationally expensive, since it is $\#P$-complete [32]. On the other hand, since channel capacity is computed as the logarithm of the number of outputs, imprecision in the model counting procedure will result only in minor variations of the computed channel capacity. Hence, it is natural to consider using approximate model counting techniques, where the precision of the result is traded for improved efficiency.

This idea has been investigated by Klebanov et al. [22]. However, in Section 5.1 we show that the approach in [22] is fundamentally incorrect, requiring a different technique.

For these reasons, in this paper we propose ApproxFlow, a new QIF computation technique to tackle problems for which precise model counting is not efficient enough. ApproxFlow is based on the ApproxMC2 tool implemented by Chakraborty et al. [12]. We show that ApproxFlow vastly outperforms the state of the art on all but a few of the benchmarks, including on many cases in which no other tool is able to provide an answer, making it the most efficient tool for QIF computation currently available. The contributions of this paper are:

- We present ApproxFlow, a technique to quantify information flow for deterministic C programs based on the approximate projected model counter ApproxMC2;
- We show that a small decrease in the approximation precision can yield large performance improvements, allowing ApproxFlow to scale to complex cases with minimal reduction in the result's usefulness;
- We show that the technique presented in [22] is incorrect due to some mistakes in its underlying theoretical results;
- We evaluate ApproxMC2 against the precise projected counter sharpCDCL on a large set of benchmarks, showing that the former generally yields orders of magnitude better performance at the cost of a small decrease in precision;
- We use ApproxFlow's improved scalability to model and compute the leakage of the code in the Heartbleed bug [1], unlike previous QIF techniques.

The rest of the paper is structured as follows. Section 2 introduces technical background and notation, and Section 3 discusses related work. We describe our technique ApproxFlow Section 4 and evaluate it in Section 5. Section 6 presents the Heartbleed case study. Section 7 provides additional discussion, and Section 8 concludes the paper.

2 Background

This section introduces the background and notations used in this paper.

Entropy and channel capacity. Let $\mathcal{X}$ be a discrete finite *sample set* and $\rho(\mathcal{X})$ a probability distribution on it, where the probability of an outcome $x \in \mathcal{X}$ is denoted $\Pr[x, \rho(\mathcal{X})]$. We omit ρ from the notation of Pr whenever ρ is clear from the context. We denote $\mathcal{U}(\mathcal{X})$ to denote a uniform distribution over $\mathcal{X}$. The *entropy* $H(\rho(\mathcal{X}))$ of a probability distribution $\rho(\mathcal{X})$, measured in bits, is defined as $H(\rho(\mathcal{X})) = -\sum_{x\in\mathcal{X}} \Pr[x] \cdot \log_2 \Pr[x]$. The *conditional entropy* $H(\rho(\mathcal{Y}|\mathcal{X}))$ of the conditional probability distribution $\rho(\mathcal{Y}|\mathcal{X})$, is defined as $H(\rho(\mathcal{Y}|\mathcal{X})) = -\sum_{y\in\mathcal{Y}} \sum_{x\in\mathcal{X}} \Pr[y|x] \cdot \log_2 \Pr[y|x]$, where $\Pr[y|x]$ denotes the probability of an outcome $y \in \mathcal{Y}$ given that an outcome $x \in \mathcal{X}$ has already occurred.

Define a *deterministic channel* D as a triple $(\mathcal{I}, \mathcal{O}, F)$ where $\mathcal{I}$ (inputs) and $\mathcal{O}$ (outputs) are discrete finite sample sets and F is a function $\mathcal{I} \to \mathcal{O}$ defining which output $o \in \mathcal{O}$ is produced for each input $i \in \mathcal{I}$. Hence, any probability distribution $\rho(\mathcal{I})$ on the input $\mathcal{I}$ induces a probability distribution $\rho(\mathcal{O})$ on the output $\mathcal{O}$ via F. The *mutual information* of a deterministic channel D for a given $\rho(\mathcal{I})$ is defined as $I(D, \rho(\mathcal{I})) = H(\rho(\mathcal{O}))$.

The *channel capacity* of D is defined as $C(D) = \max_{\rho(\mathcal{I})} I(D, \rho(\mathcal{I}))$ where the maximum is taken over all possible probability distributions $\rho(\mathcal{I})$. Note that for a deterministic channel D, it is known [16] that $C(D) = \log_2 |\{o \in \mathcal{O} \text{ s.t. } \Pr[o, \rho(\mathcal{O})] > 0\}|$ where $\rho(\mathcal{O})$ is induced by $\rho(\mathcal{I}) = \mathcal{U}(\mathcal{I})$. In particular, $C(D)$ is precisely the range of the function F.

A deterministic program can be regarded as a deterministic channel, where $\mathcal{I}$ and $\mathcal{O}$ represent the possible values for the program's inputs and outputs. In this case the channel capacity represents the maximum amount of information that can be inferred on the program's inputs by observing its outputs [17].

Model counting. *Model counting*, or #SAT, is the canonical $\#P$-complete problem, and is the counting analogue of the Boolean satisfiability (SAT) problem [32]. Let ϕ be a *SAT formula* involving variables $\mathcal{V}$, and $a \in \{\texttt{true}, \texttt{false}\}^{\mathcal{V}}$ be a Boolean valuation of $\mathcal{V}$. We say that a is a model of ϕ, denoted $a \vdash \phi$, if ϕ evaluates to `true` when substituting variables with their value in a.

The model count $\#\phi$ of ϕ is the number of valuations that satisfy ϕ:

$$\#\phi = \left|\left\{a \in \{\texttt{true}, \texttt{false}\}^{\mathcal{V}} \mid a \vdash \phi\right\}\right| \ .$$

We introduce the notion of *projection* in the context of model counting [3]. We consider a subset $\mathcal{S} \subseteq \mathcal{V}$ of the variables. Given a Boolean valuation a of $\mathcal{V}$, we naturally define its projection $a|_{\mathcal{S}}$ on $\mathcal{S}$ by restricting the input domain of a to $\mathcal{S}$. The projection $a|_{\mathcal{S}}$ is a Boolean valuation of $\mathcal{S}$.

The projected model count of a SAT formula ϕ on a *projection scope* $\mathcal{S}$ is the number of valuations of $\mathcal{S}$ that can be extended into a model of ϕ:

$$\#\phi_{\mathcal{S}} = \left|\left\{a_{\mathcal{S}} \in \{\texttt{true}, \texttt{false}\}^{\mathcal{S}} \mid \exists a \in \{\texttt{true}, \texttt{false}\}^{\mathcal{V}} a|_{\mathcal{S}} = a_{\mathcal{S}} \wedge a \vdash \phi\right\}\right| \ .$$

Approximate projected model counting with ApproxMC2 Approximate projected model counting [12] refers to the problem of finding an estimate on the projected model count of a SAT formula ϕ onto a subset $\mathcal{S}$ of the variables, as opposed to precise number.

We present the core ideas behind the ApproxMC2 tool used in this paper, and refer the reader to [12, 11] for a full exposition. ApproxMC2 is a *Karp-Luby* (or (ε, δ)) counter [20], which obtains an estimate $\widehat{\#\phi}$ on $\#\phi$ that falls within a factor $1 + \varepsilon$ of $\#\phi$ with a probability of $1 - \delta$, i.e., given a tolerance ε and a probability δ it holds that

$$\Pr\left[(1+\varepsilon)^{-1} \cdot (\#\phi) \leq \widehat{\#\phi} \leq (1+\varepsilon) \cdot (\#\phi)\right] \geq 1 - \delta \ .$$

ApproxMC2 works by randomly partitioning the set of possible models of the SAT formula ϕ projected onto $\mathcal{S} \subseteq \mathcal{V}$ (denoted as $\phi_{\mathcal{S}}$), into roughly equal buckets, performing model counting on this single bucket, and returning this count, multiplied by the number of buckets, as the approximation of the exact projected model count $\#\phi_S$. The partitioning into buckets of roughly equal size is key, and is done using an approach based on *r-wise independent hash functions* [6], adding special randomized *XOR constraints* to the SAT formula. If these randomly-chosen buckets are "too big," the number of buckets is doubled and the procedure is repeated with accordingly smaller buckets.

For the reader's convenience, we present a description of ApproxMC2, the algorithm from [12], in Algorithm 1. We note that the algorithm has a chance to

Algorithm 1: ApproxMC2

Input : A SAT formula ϕ with $|\phi|$ SAT variables $x_1, ..., x_{|\phi|}$
Input : A projection scope $S \subseteq \{x_1, ..., x_{|\phi|}\}$
Output: An estimate of $\#\phi_S$, the number of models of ϕ projected onto S

1 $p \leftarrow 1 + 9.84 \cdot (1 + \frac{\epsilon}{1+\epsilon}) \cdot (1 + \frac{1}{\epsilon})^2$ // pivot value p
2 $b \leftarrow \min(p, \#\phi_S)$ // return p as soon as $\geq p$ models of $\#\phi_S$ found
3 **if** $b < p$ **then**
4   **return** b
5 cells $\leftarrow 2$ // Number of cells
6 $C \leftarrow [\,]$ // Empty list
7 **for** $i \leftarrow 1$ ***to*** $\lceil 17 \cdot \log_2(\frac{3}{\delta}) \rceil$ **do**
8   Choose h at random from $H_{xor}(|S|, |S| - 1)$ // Random hash function
9   Choose α at random from $\{0, 1\}^{|S|-1}$
10     $\phi' \leftarrow \phi \wedge h(S) = \alpha$ // Add random XOR constraint to ϕ
11     $b' \leftarrow \min(p, \#\phi'_S)$
12   **if** $b' \geq p$ **then**
13   cells $\leftarrow \bot$
14   models $\leftarrow \bot$
15   **if** cells $\neq \bot$ **then**
16         $m \leftarrow \log_2$ LogSATSearch($\phi, S, h, \alpha, p, \log_2$ cells)
17         $\phi'' \leftarrow \phi \wedge h^{(m)}(S) = \alpha^{(m)}$ // Add XOR constraints to ϕ
18   models $\leftarrow \min(p, \#\phi''_S)$
19   AppendToList(C, cells $\cdot$ models)
20 **return** $\tilde{C}$ // Median of C

fail to return anything at line 4, when it returns $\bot$. By repeating the algorithm a sufficiently large number of times, we can obtain the desired probability $1-\delta$ that it will succeed. In line 16, the invocation of LogSATSearch refers to a procedure to obtain good values for m. This is beyond the scope of this paper, and we refer to [12] for details. In lines 2 , 11 , and 18, the minimum is computed using a SAT solver which iteratively finds up to p models. Note that this step does not require the usage of a model counter. Thus, the precise model count is not typically computed at these points, unless the formula (augmented with any XOR constraints) has become constrained (small) enough to have p or fewer models.

We emphasize that ApproxMC2 allows us control the tolerance ε. We will show in Section 4.3 how reducing the tolerance can significantly improve the computation time.

3 Related Work

This section presents a short review of work that is related to this paper.

3.1 Quantitative Information Flow

Prior work on QIF has largely followed the paradigm of characterizing the set of a program's outputs. We classify related work into two categories: those which measure channel capacity, and those that measure other kinds of entropy. We make a note that some work in channel capacity formulates their problem in terms of min-entropy, but it is known [25] that for *deterministic* channels, min-entropy and channel capacity are equivalent. In addition, because much work in QIF considers conditional entropy, we remark that channel capacity corresponds to minimizing the conditional entropy of the output given the input [17]. This is easy to see, as (adopting the definitions and notation from Section 2), $C(D) = \max_{i \in \mathcal{I}} I(D) = \max_{i \in \mathcal{I}}(H(\rho(\mathcal{O})) - H(\rho(\mathcal{O}|\mathcal{I}))$. To maximize $I(D)$, $H(\rho(\mathcal{O}|\mathcal{I}))$ must be minimized.

Channel capacity. Meng and Smith [25] present a method to obtain empirically good upper bounds on the channel capacity of various small synthetic example programs, also contributing to standardizing a set of QIF benchmark programs. In [21], Klebanov et al. show how to obtain precise measurements of the channel capacity (alongside the conditional Shannon entropy) for a number of programs, including the benchmarks from [25], in addition to a number of small synthetic programs and two examples of real C code on the order of magnitude of 100 lines. In [26], Newsome et al. present a compound approach to obtain precise channel capacity measurements for a set of small, synthetic benchmark programs, and very coarse approximations to large, real-world programs up to a million lines. In [31], Val et al. present a way to measure the channel capacity for a number of benchmarks both synthetic and real, showing how to scale to programs up to thousands of lines of code. McCamant and Ernst [24] use a coarse upper-bounding approach for channel capacity based on network flows, showing how to scale to hundreds of thousands of lines of real code and contributing smaller case studies as benchmarks. Phan and Malacaria [27] present a method that is able to analyze and compute upper bounds on the channel capacity for C implementations of several well-known protocols, as well as three few-hundred-line case studies including parts of the Linux kernel. While some of the above work has demonstrated that generating SAT formulas is possible even for large programs, complex program structures such as pointers often result in SAT formulas that are too difficult for model counting. In addition, the various approaches have occupied static points on the precision vs. scale relation, unable to vary precision to obtain significant speedups.

Other QIF measures. In [34], Weigl presents a tool sharpPI, which implements different search heuristics for model counting, applying it to the measurement of Shannon entropy and presenting results for a small, scalable synthetic C program. In [8], Biondi et al. present a technique, implemented in the QUAIL tool [10, 9], to measure Shannon entropy for a number of scalable case studies expressed in a simple imperative language. More recently, Fremont et al. [19] present MaxCount, a novel approximate QIF method effective at finding leaks in programs, with increasing efficacy as the relative size of the leaks increase. In [5], Backes et al. present a technique to analyze small, synthetic programs with respect to various information-theoretic measures. These techniques do not compute the channel capacity, and therefore they are not comparable with our approach. We note that, as a special case of QIF (checking for the existence of a non-zero flow), *qualitative* information flow has been demonstrated to scale to large program sizes, and confirm bugs in real software such as the OpenSSL Heartbleed bug [23]. However, qualitative information flow does not attempt to measure the *amount* of information flowing through a program, and therefore cannot be directly compared to our work.

Most recently, Biondi et al. present HyLeak [7], a tool based on a combination of channel matrix computation and simulation to compute channel capacity, among other information-theoretic measures.

3.2 Projected Model Counting

Projected model counting is a problem that arises naturally in QIF measurement [26, 21, 31, 27].

In [34], Weigl presents an approach to projected model counting used as part of a QIF measurement technique, implementing several different search heuristics to guide the model counting. In [31], Val et al. present SharpSubSAT, a simple projected model counter as part of a toolchain for measuring channel capacity, which handles projection by removing variables from the formula that are not part of the projection subset. The projected counter SharpCDCL [21] uses a similar technique based on the state-of-the-art model counter sharpSAT. SharpCDCL is in fact the current state-of-the-art tool in projected model counting.

Still, precise model counting often cannot scale to larger problem sizes, prompting the need for approximate methods. Work in approximate model counting has fallen into three categories: counters that provide no theoretical guarantees but empirically yield good estimates on the true count, counters providing a count that represents an upper (or lower) bound on the exact count, and counters that provide an interval within which the exact count falls ((ε, δ)-counters). We are especially interested in these (ε, δ)-counters, for the theoretical guarantees they provide, and for the promise of trading precision for running time. In addition, there are (to the best of our knowledge) no *projected* approximate model counters that fall outside this category.

Klebanov et al. [22] present a counter based on ApproxMC2 [11], with scalability to 10^5 variables and 10^6 clauses. However, as shown in Section 1, this particular counter has some theoretical mistakes, and we cannot consider it among the state of the art. In [12], the authors present a counter based on the one from [11], and demonstrate scalability to formulas with 10^5 variables and 10^6 clauses. Indeed, the counter from [12] is among the state-of-the-art (ε, δ)-counters with projection capabilities.

Recently, Fremont et al. [19] have presented the Maximum Model Counting technique to compute approximate subset model counts, a novel technique based on a partitioning scheme inspired by [12], and using the same underlying algorithm (ApproxMC2) as we do in our technique. In their approach, the effectiveness of the algorithm increases with the number of solutions.

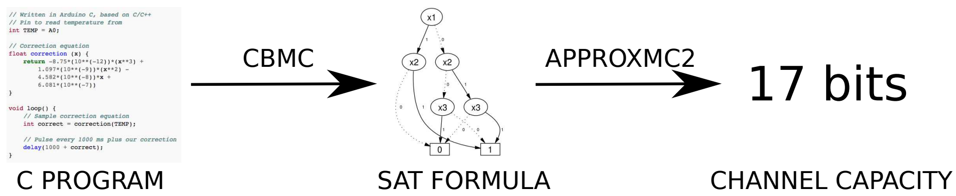

Fig. 1. A high-level view of ApproxFlow's toolchain.

4 Channel Capacity Estimation with ApproxFlow

In this section, we describe ApproxFlow[5], our technique for estimating the channel capacity of a program to a given precision, with the intention of flagging suspicious parts to a developer applying the tool on the program. We restrict ourselves to *deterministic* programs, consistent with our background in Section 2. A high-level view of our approach can be seen in Figure 1. ApproxFlow takes as input a program, and passes it to a model checker to generate a SAT formula representing the program. The SAT formula must be annotated with a projection scope, which is the subset of the variables in the formula that correspond to the original program variables to which we wish to measure channel capacity. We consider these the "output" variables (although they can be SAT variables corresponding to program variables anywhere in the program), while the SAT variables corresponding to the original program inputs are not projected upon or constrained in any way. ApproxFlow then passes this annotated SAT formula to a projection-capable approximate model counter in order to obtain an approximation on the number of models of the formula, projected onto the projection scope. Finally, ApproxFlow takes the logarithm in base 2 to obtain our final measurement – an approximation of the channel capacity of the program. The following subsections provide details on each part of the toolchain.

4.1 Program to SAT Formula

ApproxFlow takes as input a deterministic C program, and uses the model checker CBMC [15] to obtain an annotated SAT formula that represents the original program. The C program may be optionally annotated by the user to specify a given program location or set of program variables to which to measure the channel capacity. In practice, this program annotation is specified using CBMC's assertion facilities. The user can use the assertion `__CPROVER_assert(0,"");` to specify where the formula should be computed. If this annotation is not provided by the user, ApproxFlow automatically converts the program into an equivalent program where all functions in the program have a single return point at the end of each function, and the annotation is automatically placed immediately before this single return point.

[5] ApproxFlow is publicly available at `https://github.com/approxflow/approxflow`.

CBMC performs bounded model checking on the program[6], and outputs a SAT formula in conjunctive normal form (CNF) that represents the constraints on the variables induced by the program. This model checking step is subject to the following limitations: 1) loop unwinding is bounded to a specified depth (always set high enough in our experiments to capture the full behaviour of the program), and 2) the set of possible values for pointers is overapproximated.

For a fuller treatment of the effect of bounded loop unwinding on (precise) channel capacity computation, we refer the reader to [31], but we give a brief treatment of the topic. For some programs, such as server software which includes infinite loops by design, loop unwinding limits the scope of the analysis, and underapproximates program behaviour. Consequently, the channel capacity is also underapproximated. However, it is often the case that an output variable to which we measure leakage already achieves maximum leakage after only a few iterations. In addition, many loops are executed for only a few iterations, and a bound such as 32 (our default) is more than enough to capture the loop's full behaviour. As our goal is *approximate* channel capacity measurement, we argue that our approach is less sensitive than precise approaches to the further approximation induced by bounding the loop unwinding. A similar argument can be made for the *overapproximation* caused by CBMC's conservative pointer analysis, and we again refer the reader to [31] for a discussion in the context of information flow. Both issues are orthogonal to our contributions, as they result directly from CBMC's limitations in performing a more precise analysis; improvements in model checking and formula generation would benefit us directly.

Additionally, CBMC annotates the SAT formula with comments that specify which boolean variables in the SAT formula correspond to the original program variables. In this way, we are able to obtain a SAT formula from CBMC that is annotated with our desired projection scope, which may be then passed to an approximate model counter in order to obtain the number of models of the formula projected onto the specified variables.

4.2 SAT Formula to Channel Capacity

In the second step of our approach, we take as input an annotated SAT formula obtained from the model checker and use a projection-capable approximate model counter to obtain an estimate of the number of models of the projected formula. Specifically, we use an improved implementation of a state-of-the-art approximate model counter ApproxMC2 [12] by Mate Soos and Kuldeep Meel, which is pending publication. For the remainder of the paper, whenever we refer to ApproxMC2, we are actually referring to this improved version.

ApproxMC2 takes a SAT formula in conjunctive normal form (CNF), specified in the DIMACS CNF format [4], with the projection scope specified by special comments in the file. ApproxMC2 provides an approximate number of models of this projected formula within a specified tolerance, with high probability.

[6] We do not discuss model checking in this paper. For a treatment of model checking, please consult [14].

While ApproxMC2 is a state-of-the-art approximate counter, it does have some limitations even when compared to precise tools. ApproxMC2 has significant overhead due to the requirement of adding XOR constraints, which tends to make it perform more poorly in terms of running-time on smaller problems relative to other counters. In addition, ApproxMC2's expected runtime is higher when compared to other counters when it is used to solve formulas that are *dense* in their solution space – that is, formulas which have a large number of models in relation to their formula size (in number of variables). In practice, these limitations are not usually a problem compared to other available counters (precise or approximate). Full details may be found in [12].

Finally, ApproxFlow takes the logarithm in base 2 of this estimated model count in order to obtain an estimation of the channel capacity of the program. Somewhat unique to this problem, it is worth nothing that taking the logarithm of the approximate count exponentially squishes the error in the estimate. In other words, a fairly coarse approximation on the model count can yield good probabilistic bounds on the channel capacity estimate.

4.3 Performance-Precision Trade-off

Using an approximate method naturally leads to a trade-off between precision and performance. Because ApproxMC2 is able to trade a lower precision for a shorter running time, we can choose a trade-off point on the side of shorter running time when a close approximation is not essential (for instance, when an approximate lower bound is the desired outcome, as would be desired when enforcing k-bit policies [26]).

We evaluate this trade-off for ApproxMC2 on the AppleTalk Linux driver benchmark, ddp.pp (discussed in more detail in Section 5.3). This benchmark exhibits a large enough channel capacity (128 bits) such that a result with a few bits of imprecision is still useful. In addition, it is long-running enough (roughly 20 seconds on our machine, detailed in Section 5) to be largely immune to small variations in time resulting from background CPU usage, making it a good candidate for trading precision for performance.

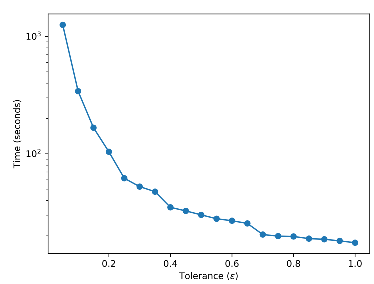

Fig. 2. Precision-time relationship for ApproxMC2. Time (in seconds) is on the vertical axis, and the precision (or *tolerance*) parameter ε is shown on the horizontal axis. Larger values of ε represent more relaxed precision guarantees. All measurements taken were for the preprocessed AppleTalk case (ddp.pp.cnf) from Section 5 with probability $(1 - \delta) = 0.8$.

Figure 2 shows the relationship between precision ε and running time for $0.05 \leq \varepsilon \leq 1$, with a fixed δ of 0.2 (the default value). As time is plotted on a logarithmic scale, we can see that as soon as we are able to relax the precision/tolerance of the method by increasing epsilon, we gain a dramatic speedup. Since channel capacity is computed as the logarithm of the number of models, the worst case, $\varepsilon = 1$, represents only a single bit of imprecision, yet we observe a speedup factor of 2-3 orders of magnitude over the $\varepsilon = 0.05$ case, which represents an imprecision of $\log_2(1.05) \approx 0.070$ bits. Interestingly, we observed that in all cases, the reported information flow was 128 bits. In practice, the trade-off is even better based on these empirical observations than expected from the theoretical guarantees.

5 Evaluation

In this section, we present an experimental evaluation of ApproxFlow compared to the state-of-the art precise channel capacity measurement tools. We ran all experiments on an Oracle VirtualBox virtual machine with 1 CPU and 8GB of RAM running Linux Mint 18.1 hosted on a Windows 10 machine with a quad-core Intel Core i7 2.9GHz CPU and 16GB of RAM.

5.1 Problems in Klebanov et al. [22]

The usage of approximate model counting to determine channel capacity was previously explored by Klebanov et al. [22]. However, we have been unable to replicate the results in [22]. After further investigation, we have concluded that our inability to replicate such results depends on the fact that the theoretical claims and proofs presented in [22] are incorrect.

The main result of [22] is presented in Theorem 2.12. The theorem aims to show that the algorithm described in the paper terminates with high probability, returning an estimate on the approximate model count. Unfortunately, as result of a mistake in the proof, the probability of termination is overestimated, and the presented algorithm appears to be more effective than it actually is at approximating the true number of solutions. In particular, the proof of this theorem hinges on the following claim (adopting the notation from [22]):

> We now show that there is at least one iteration of the loop (indexed by $m = m'$) such that with a probability of at least $1 - e^{\lfloor -r/2 \rfloor}$ the following is true: the exit condition $c \leq pivot$ holds and the return value $2^{m'} \cdot |\phi_h| \in [(1-\varepsilon)|\phi|, (1+\varepsilon)|\phi|]$

The authors then proceed to prove the above claim and treat it as sufficient for the proof of Theorem 2.12. However, this is incorrect since the above claim is not a sufficient condition for Theorem 2.12. To this end, let us define the event T_i as condition $c \leq pivot$ holds for iteration $m = i$ and the event U_i as $2^i \cdot |\phi_h| \in [(1-\varepsilon)|\phi|, (1+\varepsilon)|\phi|]$. Theorem 2.12 seeks to bound from below the probability of the event S, where $S = \cup_{i=1}^{n}((\cap_{j=1}^{i-1}(\bar{T_j})) \cap T_i \cap U_i$. Note that, $\Pr[S] \geq \Pr[T_i \cap U_i]$ does not necessarily hold for all i. Therefore, demonstrating that there exists $m = m'$ such that $\Pr[T_m \wedge U_m] \geq 1 - e^{\lfloor -r/2 \rfloor}$ is not sufficient to support the claim in Theorem 2.12.

In addition, there is an error in the statement of Theorem 2.6. The authors have written the upper bound on the probability as $e^{-\lfloor r/2 \rfloor}$, instead of $e^{\lfloor -r/2 \rfloor}$ [29, Theorem 5]. The authors conclude that the value for *pivot* reported in [11] can be made smaller, chosen as the value reported in Algorithm 3, Line 2 and shown in Table 1. However, these conclusions are supported by Lemma 2.13, which depends on Theorem 2.6 and the incorrect bound on the probability. As a result, the reported precision of the algorithm is overestimated compared to the true precision.

5.2 Comparison to Precise Channel Capacity

We compare ApproxFlow with the SharpCDCL-based technique proposed by Klebanov et al. [21]. Both techniques follow the same steps: 1) generating a SAT formula from a program using CBMC [15], 2) specifying a projection scope, and 3) performing projected model counting (precise or approximate). We compare only the step where ApproxFlow differs from Klebanov et al.'s approach, namely projected model counting. For this comparison, we feed a SAT formula to both tools, along with a projection scope extracted from a C program. If SAT formulas are directly available from the existing benchmarks, we reuse those formulas, otherwise we generate them with CBMC.

To produce the SAT formulas we use a 32-bit CBMC version 5.6 with the arguments `--32 --dimacs --function function-name --unwind loop-bound`. The parameter `function-name` specifies the function containing the variables for which we want to measure channel capacity. The parameter `loop-bound` is the loop unrolling depth, set to 32. We insert projection scopes corresponding to the SAT variables in the formats required by sharpCDCL and ApproxMC2.

Finally, we run sharpCDCL and ApproxMC2 each with a timeout of 2 hours, unless otherwise specified. We measure the running time and the number of models reported by each tool. As explained earlier, the base-2 logarithm of the number of models gives us the channel capacity. If sharpCDCL times out, it reports a lower bound on the number of models it found, while ApproxMC2 does not currently have this feature implemented. Consequently, we report a lower bound only for sharpCDCL, when applicable. We run sharpCDCL with arguments `-countMode=2 -projection=projection-scope`, where `projection-scope` refers to a file containing the projection variables, and a `countMode` of 2 tells sharpCDCL to perform model counting, rather than just SAT-solving. We ran ApproxMC2 with no arguments, as the projection scope is specified as comments in the SAT formula file. The default tolerance ε for ApproxMC2 is 0.8 (~ 0.8 bits of error), and the default confidence is 80% ($\delta = 0.2$).

While we recognize that a large number of runs for each experiment would be ideal for statistical evidence with respect to running time, many of our experiments are long-running and doing this was not feasible. Therefore, our figures represent the results of a single invocation of each tool.

Table 1. Leakage reported by ApproxMC2 and sharpCDCL as a number of bits, relative error (as a percentage) in number of bits of channel capacity, running times for each tool, and speedup factor observed when running ApproxMC2 instead of sharpCDCL for several benchmarks. Negative entries represent slowdown factors. Speedup entries marked as — represent entries for which at least one of the tools completed too quickly for the precision of our timing tool (and at least one reported 0.00s). Entries marked with **error** represent values for which sharpCDCL terminated with an error, and could not produce a value for the model count. We note that in many cases, only ApproxMC2 was able to complete, with bolded entries representing cases in which both tools ran to completion. We note that ApproxMC2 never produced an error.

Benchmarks from [26, 5, 25, 21]

Experiment name	sharpCDCL leakage	ApproxMC2 leakage	Relative error	sharpCDCL time	ApproxMC2 time	Speedup factor
e-purse	**5.00**	**5.00**	**0%**	**0.06**	**0.28**	**-4.67**
pw-checker	**1.00**	**1.00**	**0%**	**0.00**	**0.00**	—
sum-query	>22.49	32.00	*	t/o	0.87	*
10random	**3.32**	**3.32**	**0%**	**0.00**	**0.00**	—
bsearch16	**16.00**	**16.00**	**0%**	**3.40**	**0.49**	**6.90**
bsearch32	>22.87	32.00	*	t/o	2.13	*
mix-dupl	**16.00**	**16.00**	**0%**	**5.91**	**0.20**	**29.60**
sum32	>22.48	32.00	*	t/o	0.89	*
illustr.	**4.09**	**4.09**	**0%**	**0.00**	**0.01**	—
mask-cpy	**16.00**	**16.00**	**0%**	**6.02**	**0.20**	**30.1**
sanity-1	>22.82	31.04	*	t/o	0.94	*
sanity-2	>22.92	31.00	*	t/o	1.07	*
check-cpy	>22.51	32.00	*	t/o	0.88	*
copy	>22.49	32.00	*	t/o	0.84	*
div-by-2	>22.79	31.00	*	t/o	1.06	*
implicit	**>2.81**	**2.81**	**0%**	**0.00**	**0.01**	—
mul-by-2	>22.46	31.00	*	t/o	0.89	*
popcnt	**5.04**	**5.04**	**0%**	**0.00**	**0.01**	—
simp-mask	**8.00**	**8.00**	**0%**	**0.00**	**0.05**	—
switch	**4.25**	**4.25**	**0%**	**0.00**	**0.00**	—
tbl-lookup	>22.45	32.00	*	t/o	0.88	*

5.3 Benchmarks

Several benchmarks have become accepted in the QIF literature. Tables 1 to 3 show the relative error and speedup factor of running ApproxMC2 instead of sharpCDCL on these benchmarks. As no SAT formulas were openly available for many of these, we wrote C implementations from the descriptions of the specified benchmarks in their respective papers, and obtained SAT formulas using CBMC as previously described.

Table 1 ApproxMC2 and sharpCDCL on the benchmarks presented in [26, 5, 25, 21]. When both ApproxMC2 and sharpCDCL terminate before the time out, the reported model count is identical, therefore ApproxMC2 has relative error of zero. In the cases when sharpCDCL times out after two hours, the lower-bound channel capacity reported by sharpCDCL ranges from 22 to 23 bits, even when the actual result is larger. On these benchmarks, ApproxMC2 is not much slower than sharpCDCL, and always reports the exact result. The converse tells a different tale, with sharpCDCL often timing out after 2 hours, and providing only a coarse lower bound in these cases. It is also somewhat surprising that sharpCDCL times out on SAT formulas resulting from some simple programs, such as divide-by-2.

Table 2. Leakage reported by ApproxMC2 and sharpCDCL as a number of bits, relative error (as a percentage) in number of bits of channel capacity, running times for each tool, and speedup factor observed when running ApproxMC2 instead of sharpCDCL for several benchmarks. Negative entries represent slowdown factors. Speedup entries marked as — represent entries for which at least one of the tools completed too quickly for the precision of our timing tool (and at least one reported 0.00s). Entries marked with error represent values for which sharpCDCL terminated with an error, and could not produce a value for the model count. We note that ApproxMC2 never produced an error, and further note that in many cases, only ApproxMC2 was able to complete, with bolded entries representing cases in which both tools ran to completion. The entry fx was run with a higher timeout (8.5 hours) instead of the usual 2 hours.

Benchmarks from [26, 5, 25, 21]

Experiment name	sharpCDCL leakage	ApproxMC2 leakage	Relative error	sharpCDCL time	ApproxMC2 time	Speedup factor
ddp	error	128.00	∗	error	23.50	∗
ddp.pp	error	128.00	∗	error	19.55	∗
popcount	**5.04**	**5.04**	**0%**	**0.00**	**0.01**	—
sanitize	**4.00**	**4.00**	**0%**	**0.00**	**0.00**	—
openssl.1	**8.00**	**8.00**	**0%**	**1.44**	**70.66**	**-49.10**
openssl.2	**16.00**	**16.00**	**0%**	**4.63**	**75.39**	**-16.30**
openssl.3	>22.24	24.00	∗	t/o	92.47	∗
openssl.4	>22.91	32.00	∗	t/o	86.32	∗
openssl.5	>23.10	40.00	∗	t/o	87.74	∗
openssl.6	error	48.00	∗	error	89.60	∗
openssl.7	error	56.00	∗	error	91.98	∗
openssl.8	error	64.00	∗	error	98.04	∗
openssl.9	error	72.00	∗	error	97.41	∗
openssl.10	error	80.00	∗	error	112.71	∗
openssl.15	error	t/o	∗	error	t/o	∗
openssl.20	error	160.00	∗	error	142.48	∗
swirl	>12.82	t/o	∗	t/o	t/o	—
10random	**3.32**	**3.32**	**0%**	**0.00**	**0.01**	—
bsearch16	**16.00**	**16.00**	**0%**	**4.16**	**0.68**	**6.12**
bsearch16.pp	**16.00**	**16.00**	**0%**	**3.73**	**0.35**	**10.70**
bsearch32	>22.79	32.00	∗	t/o	3.21	∗
bsearch32.pp	>22.90	32.00	∗	t/o	6.93	∗
fx	**16.00**	**16.00**	**0%**	**5753.42**	**7307.61**	**-1.27**
mixdup	**16.00**	**16.00**	**0%**	**8.44**	**0.22**	**38.40**
sum.32	>22.78	32.00	∗	t/o	0.98	∗

In Table 2, we present results for a set of benchmarks described in [22, 21], for which the authors kindly provided us the SAT formulas directly. As in the previous set of experiments, when both ApproxMC2 and sharpCDCL report a number of models, the numbers are identical despite ApproxMC2's fairly relaxed theoretical tolerance and confidence. In these experiments, we found that sharpCDCL sometimes incorrectly terminates before its timeout because of two kinds of error: a segmentation fault, or reporting the formula to be unsatisfiable (despite normally giving a lower bound on the number of solutions if interrupted). In addition, we witness cases in which ApproxMC2 timed out. We observe that ApproxMC2 is slower than sharpCDCL on short-running experiments (openssl.1 and openssl.2), but significantly faster on the more difficult, longer-running experiments (where sharpCDCL often times out), with the exception of fx.

In Table 3, we present results for a set of scalable case studies given in [9]. These case studies consist of two models of a Voting protocol (one based on each voter voting for a single-candidate, and one based on each voter having a

Table 3. Leakage reported by ApproxMC2 and sharpCDCL as a number of bits, relative error (as a percentage) in number of bits of channel capacity, running times for each tool, and speedup factor observed when running ApproxMC2 instead of sharpCDCL for several benchmarks. Negative entries represent slowdown factors. Entries marked with error represent values for which sharpCDCL terminated with an error, and could not produce a value for the model count (even in cases it completed within the timeout, it did not report a number of solutions). We note that ApproxMC2 never produced an error, with bolded entries representing cases in which both tools ran to completion. The entries with 0% error had a reported solution count of 0 by both tools, so we abuse notation and consider this a 0%, rather than undefined, error.

Benchmarks from [26, 5, 25, 21]

Experiment name	sharpCDCL leakage	ApproxMC2 leakage	Relative error	sharpCDCL time	ApproxMC2 time	Speedup factor
Sing.3	error	5.81	*	error	1.46	*
Sing.5	**7.62**	**7.86**	**3.15%**	**0.06**	**3.02**	**-50.30**
Sing.7	**9.63**	**9.70**	**0.73%**	**0.38**	**3.98**	**-10.50**
Sing.9	**10.97**	**11.00**	**0.27%**	**0.83**	**5.82**	**-7.01**
Rank.3	>21.00	67.17	0%	t/o	55.34	*
Rank.5	**0.00**	**0.00**	**0%**	**0.40**	**0.52**	**-1.30**
Rank.7	**0.00**	**0.00**	**0%**	**0.75**	**0.96**	**-1.28**
Rank.9	**0.00**	**0.00**	**0%**	**1.26**	**1.58**	**-1.25**

preference ranking of the candidates). These experiments have parameters that control the size of the program, and therefore of the generated SAT formula. We refer the reader to [9] for a description of the case studies and their parameters. We translated the Java code provided on the companion website of the paper into C, and generated SAT formulas with CBMC, with 16 as the bound for loop unwinding. The experiment names beginning with "Sing" represent the single candidate case from the case studies, while the experiment names beginning with "Rank" represent the preference ranking case. In both cases, we correspond cases in which ApproxMC2 is not clearly better than sharpCDCL. Although sharpCDCL produces an error or times out in two of these cases, when sharpCDCL terminates, it is between 7 and 50 times faster than ApproxMC2. We believe this is because the resulting SAT formulas are dense in the number of solutions, which is a weakness of ApproxMC2 (as we stated in Section 4.2). Nonetheless, ApproxMC2 is very precise, exhibiting relative errors ranging from 0.30% to 3.10%. The Rank entries with the number of candidates ranging from 5 to 9 represent unsatisfiable formulas, and thus have 0 solutions.

As a consequence of the errors returned by sharpCDCL and the number of benchmarks for which ApproxMC2 reported the exact count, we lack an in-depth empirical evaluation of ApproxMC2's precision. To this end, we present further relative error measurements on the SmartGrid benchmark from [9]. These benchmarks compute the leakage of private information obtained by observing the global energy consumption in a smart grid. One model computes the information about a single house, and the other computes the information about the consumption of every house. As in the Voting protocol, we can scale the benchmark by changing the values of the protocol's parameters (Case A or B), the number of houses, and (for the single-house case), the size of the house – small (S), medium (M), or large(L).

We refer the reader to [9] for the full details of these models and their parameters. We present in Table 4 the relative error percentage of ApproxMC2 with respect to sharpCDCL, on the number of bits of leakage reported. As we can see, the channel capacity reported by the tools was very close (and in many cases exactly equal) in all cases when both tools ran to completion and reported a figure, except for case A, N=36 of the global leakage experiment, where we see an "error" of 45.25% when compared to sharpCDCL. This large error results from an incorrect channel capacity measurement reported by sharpCDCL. We verified this using the exact projected model counter SharpSubSAT from [31], observing a relative error of 2.86% in the channel capacity when compared to this counter.

Table 4. Relative error (as a percentage) in the channel capacity estimation by using ApproxMC2 instead of the precise counter sharpCDCL for the SmartGrid case study from [9]. The entry marked as — represents a case in which sharpCDCL returned an error and could not report a result. The entry marked with a ⋆ represents a case in which sharpCDCL returned an incorrect channel capacity (resulting in an observed 45.25% relative error). We compared ApproxMC2 to another exact counter (SharpSubSAT [31]) which reported the correct precise value, to obtain the 2.86% figure.

		Relative error			
	Num	Single house			Global
Case	**houses**	**S**	**M**	**L**	
A	**36**	0.32%	0.32%	0.32%	2.86%⋆
A	**49**	0.00%	0.00%	0.00%	0.00%
A	**64**	0.31%	0.31%	0.31%	0.32%
B	**36**	0.20%	0.58%	0.20%	—
B	**49**	0.26%	0.26%	0.26%	0.26%
B	**64**	0.10%	0.10%	0.29%	0.10%

Finally, we remark that, in addition to being much faster in most cases while maintaining very high precision, ApproxMC2 is able to report an approximate model count in all our experiments, in contrast to the significant number of error cases reported by sharpCDCL.

Comparison to ApproxMC-P

Although the work presented in [22] suffers from the theoretical errors that we described in Section 1, we compared against the implementation of their algorithm, called ApproxMC-P. We repeated the experiments from Section 5.3 for the Voting and SmartGrid case studies, using ApproxMC-P with the cryptominisat4 [30] backend, instead of ApproxMC2. We used the same values of ϵ and δ as in Section 1, but ran with a timeout of only 5 minutes instead of 2 hours. We found that in all but 2 cases, the tool reported a spurious model count of 0 (in those two non-zero cases, ApproxMC-P reported the exact count). We also tried using the sharpCDCL backend instead of the cryptominisat4 backend, and results were similar, with most cases resulting in an error. Additionally, we also ran on other experiments described in Section 5, observing a high occurrence of 0 reported as the model count. We similarly omit these due to space constraints.

```c
1   int dtls1_process_heartbeat(SSL *s) {
2
3     unsigned char *p = &s->s3->rrec.data[0], *pl;
4     unsigned short hbtype;
5     unsigned int payload;
6     unsigned int padding = 16;
7     //...
8     hbtype = *p++;
9     n2s(p, payload);
10
11    if (1+2 + payload+16 > s->s3->rrec.length)
12      return 0; /* missing in bugged version */
13
14    if (hbtype == TLS1_HB_REQUEST) {
15      unsigned char *buffer, *bp;
16      unsigned int write_length =
17         1 + 2 + payload + padding;
18      //..
19      buffer = OPENSSL_malloc(write_length);
20      bp = buffer;
21      *bp++ = TLS1_HB_RESPONSE;
22      s2n(payload, bp);
23      memcpy(bp, pl, payload);
24      //send buffer ...
25    }
26  }
```

(a)

```c
1   int dtls1_process_heartbeat(char* input_msg,
2                               int   msg_len){
3     char *p = input_msg;
4     unsigned short hbtype;
5     unsigned int payload ;
6     unsigned int padding = 0; // ignore padding
7     hbtype = *p;
8     p++;
9     n2s(p,payload);
10
11    // only present in model for correct version
12    __CPROVER_assume(1 + 2 + payload <= msg_len);
13
14    // we model only the if true branch
15    unsigned char buffer[3 + MAX_PAYLOAD_SIZE];
16    unsigned char *bp;
17
18    set_to_zero(buffer, 3 + MAX_PAYLOAD_SIZE);
19    bp = buffer;
20    *bp = TLS1_HB_RESPONSE;
21    bp++;
22    s2n(payload,bp);
23    memcpy_emul(bp,p,payload);
24
25    return 0;
26  }
```

(b)

Fig. 3. Code model for the Heartbleed bug. a) Simplified fragment of code from `ssl/d1_both.c` in OpenSSL 1.0.1f. b) Model for analysis.

6 Case Study: Heartbleed Bug

We present a case study for our technique based on the Hearbleed OpenSSL bug [1]. We show that ApproxFlow can handle the complexity required to detect the bug, in contrast to the state of the art of precise QIF.

The Heartbleed bug. The Heartbleed bug [1] is a vulnerability in the OpenSSL implementation of the Heartbeat extension of TLS and DTLS [2]. It was introduced in the OpenSSL code in 2012, and discovered and patched between March and April 2014. It has been estimated that at discovery time between 24% and 55% of the HTTPS servers in the Alexa Top 1 Million list were vulnerable to it [18]. The fact that Heartbleed went unnoticed for 2 years led the security development community to ask why the automated techniques used to scan the OpenSSL code for vulnerabilities did not detect it earlier, and which static and dynamic techniques could be expected to find bugs similar to Heartbleed [28, 33, 35]. We show how QIF can be used to model and detect the Heartbleed bug.

Fundamentally, the bug consists of a buffer over-read on a `memcpy()` function call in the Heartbeat implementation in OpenSSL, specifically in function `dtls1_process_heartbeat()` of file `d1_both.c`. Figure 3 (a) presents a fragment of the function. To verify that a server is still functional, the Heartbeat protocol has the client ask the server to reply with a specific word. In the OpenSSL implementation, the chosen word and its length are under the control of the client. The length of the word is encoded in the first bytes of the message. The pointer `p` is set at the start of the message from the client passed to the function via the `SSL` structure in argument (line 3). First, the function decodes the type and length of the message and stores it in the `payload` variable (line 8-9). In the vulnerable version, the checking of payload (line 11) is absent, which allows the client to specify a word length greater than the actual length of the sent word. Next, the function allocates a buffer large enough to store the answer message for the client (line 19). A call to memcpy() (line 23) fills that buffer with the input word and, if the value of payload is greater than the length of the input word, the content of the memory after the input word. Finally, buffer is sent back to the client. With the bugged version, the client can obtain restricted kernel memory, which they can use to infer privileged information about the server, e.g. the server's private key.

Modeling the bug. We had to rewrite OpenSSL C code in a different form due to limitations in the currently-available tools that can produce SAT formulas from C code. This modelling step is not inherent in our approach, and may largely disappear in the long term as SAT-formula-generation tools mature. Generating the formulas from C code is out of the scope of this paper, and we rely on CBMC to perform this transformation. Therefore, the code that we actually analyzed is a model of the real code – one that CBMC can handle.

Our model is presented in Fig. 3 (b). Our goal is to compare the channel capacity of the input_msg and buffer arrays. Since calls to malloc are not well-supported by CBMC, we statically allocate the array (line 15). By default, CBMC considers that unassigned values are unconstrained, therefore we set each cell of buffer to zero with the set_to_zero macro on line 16. We then fill the buffer as in the original function, but instead of calling memcpy(), we invoke on line 21 a macro memcpy_emul that uses a loop to copy the values.

In order to statically set the size of buffer, we need to know the maximum value taken by the variable payload. This variable is encoded by 16 bits. However, we restrict it by adding CBMC constraints on input_msg so that we choose the number of bits. The constant MAX_PAYLOAD_SIZE is set accordingly. For the experiments, we restrict the value of payload to be encoded by 4 bits, which corresponds to a message of at most 15 bytes. We set the message length to 1 byte, as an attacker would do to maximize the amount of information obtained from the memory.

Due to another limitation in CBMC, we were not able to analyze the if-error-then-return idiom, replacing it with a CBMC assumption negating the condition of the if statement. Similarly, we only modeled the true branch of the second conditional.

Results. We first analyze the model of the vulnerable version, that is without the CBMC assumption about payload on line 12. Executing CBMC on the model in Fig. 3 (b) produces a SAT formula with 39272 clauses in less than 1 second. Since our memcpy is implemented by a loop on payload, we set the bounds on the loop to 260 (instead of 32 as in the benchmarks from Section 5), a figure chosen due to CBMC limitations.

First, we measure the channel capacity to input_msg. Both sharpCDCL and ApproxMC2 terminate in less than a second and return 12 bits, which correspond to 4 bits to encode the size of the message and 1 byte for the message itself.

We then measure the channel capacity to `buffer`. The sharpCDCL tool times out after 2 hours of trying to count the models in the formula. On the other hand, the ApproxMC2 tool provides an approximate channel capacity of 15 bytes in 25 seconds. Since the channel capacity to `buffer` is much more than the one to `input_msg`, there is a suspicious leak of approximately 14 bytes of information. By reducing the confidence to 50% ($\delta = 0.5$), ApproxMC2 returns 15.1 bytes in 2 seconds. After such analysis, a programmer could investigate why this leakage is so high and possibly discover the Heartbleed bug.

When adding the CBMC assumption representing the patch to fix the bug on line 12, the leakage of `buffer` is down to about 1 byte (257 models) and both sharpCDCL and ApproxMC2 terminate in less than a second. ApproxMC2 reports 264 models. This leakage value indicates that, as expected, the buffer actually transmits one byte of information and that the patch successfully removed the suspicious leak.

7 Discussion and Future Work

In this section, we discuss the broader meaning of our approach, its limitations, and provide discourse on the results of the evaluation in Section 5, as well as discussion on future directions.

We showed in Section 5 that an approximate approach can provide a large increase in performance at the cost of a small amount of precision, especially as problem sizes increase. A major strength of ApproxFlow is its ability to trade efficiency for precision simply by varying the tolerance parameter ε. The ability to relax the precision to a desired level can yield practical results in many cases. Consider a program meant to return a value from a small set of return codes. The corresponding leakage might be only 1 or 2 bits. In this case, a coarse approximation would be sufficient; an observation of *approximately* 10 bits is just as practically significant as an observation of *precisely* 10 bits – both would mark the program as suspicious, prompting further analysis.

In Section 6, we showed that with ApproxFlow, we can perform a largely automated analysis which is potentially useful in discovering, or confirming, bugs in real software. Nonetheless, we recognize the need for improvements to the technique before we can realize a fully-automated and practically useful bug-finding tool. As explained in Section 6, limitations in CBMC force us to analyze manually-simplified versions of some programs.

A possible improvement to formula generation would be to pursue source code in a language easier to analyze than C. Higher-level languages such as Java or C# present easier analysis for model checkers and symbolic execution engines, because of features such as stronger type-checking. While C is arguably still the most relevant language for targeting security bugs, it is perhaps too ambitious a target for current formula generation techniques. Since the formula generation is decoupled from the model counting, it would be interesting to study the effectiveness of our overall approach for Java or C# source code, using a tool such as Java PathFinder as its formula generation engine.

In [26], Newsome et al. present the use of channel capacity measurement as a way to enforce k-bit policies, which are policies of the form *"the program leaks no more than k bits of information from its inputs to its outputs."* Such policies may be used as an aid to a developer looking for security issues in source code – as soon as "too many" bits are found, the offending part of the program can be flagged as suspicious. This is a natural use case for approximation, as the lower bound is often already a fuzzy quantity, and a choice for the value of k may be somewhat arbitrary. As a future direction, it would be interesting to use an approximate *lower bounding* projected model counter, and observe its efficacy compared to ApproxMC2 for enforcing k-bit policies. This use case, for example, gives *quantitative* information flow techniques (such as our method) a distinct advantage over qualitative ones, which do not reason about the *size* of the flow.

Finally, it would be illuminating to compare our technique to the MaxCount tool presented by Fremont et al. [19]. Using the underlying approximate counting algorithm they present in the place of ApproxMC2, we could study how sensitive ApproxFlow is to the choice of counting algorithm. We expect that an approach based on MaxCount might be more effective than our own for large leaks (relative to the formula size), but not for small leaks. Perhaps a combination of the two counting algorithms would be the most effective in practice.

8 Conclusions

We have presented ApproxFlow, a technique leveraging approximate model counting to measure the approximate channel capacity of deterministic C programs, showing it to be among the most efficient currently-available techniques for QIF computation. The necessity of such a technique arises from both theoretical errors and practical limitations in some of the prior work that applied approximate model counting to channel capacity measurement.

ApproxFlow takes a program, performs model checking to produce a formula which represents the program, and leverages approximate projected model counting in order to obtain an approximation of the program's channel capacity. We show how ApproxFlow is more efficient than state-of-the-art techniques on a number of benchmarks, with graceful degradation in the relatively few cases when it's less efficient. In particular, on many benchmarks, we show that ApproxFlow can estimate the information flow while precise tools cannot, or otherwise obtain significant speedups while maintaining high empirical precision, and exhibiting much smaller slowdown factors when ApproxFlow is slower.

In addition, we present a new case study based on the famous OpenSSL Heartbleed bug that showcases the power of our technique. While analysis with state-of-the-art precise tools times out after 2 hours, ApproxFlow obtains the channel capacity in only 25 seconds.

Our technique opens up the possibility of automatically detecting channel capacity for larger programs than previously possible, representing a step towards automatic vulnerability detection using QIF.

References

1. CVE-2014-0160 "Heartbleed". `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`. Accessed: 2017-04-03.
2. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. `https://tools.ietf.org/html/rfc6520`. Accessed: 2017-04-03.
3. R. A. Aziz, G. Chu, C. J. Muise, and P. J. Stuckey. #∃SAT: Projected model counting. In *Theory and Applications of Satisfiability Testing, SAT 2015, 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, pages 121–137, 2015.
4. D. Babic. Satisfiability Suggested Format. Technical report, 015 1993.
5. M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 141–153, 2009.
6. M. Bellare, O. Goldreich, and E. Petrank. Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation*, 163(2):510–526, 2000.
7. F. Biondi, Y. Kawamoto, A. Legay, and L. Traonouez. Hyleak: Hybrid analysis tool for information leakage. In D. D'Souza and K. N. Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 156–163. Springer, 2017.
8. F. Biondi, A. Legay, P. Malacaria, and A. Wasowski. Quantifying information leakage of randomized protocols. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 68–87, 2013.
9. F. Biondi, A. Legay, and J. Quilbeuf. Comparative analysis of leakage tools on scalable case studies. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*, pages 263–281, 2015.
10. F. Biondi, A. Legay, L. Traonouez, and A. Wasowski. QUAIL: A quantitative security analyzer for imperative code. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 702–707, 2013.
11. S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, pages 200–216, 2013.
12. S. Chakraborty, K. S. Meel, and M. Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 3569–3576, 2016.
13. D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. *Electr. Notes Theor. Comput. Sci.*, 59(3):238–251, 2001.
14. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2001.
15. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2004, Barcelona, Spain, March 29 - April 2, 2004*, pages 168–176, 2004.

16. T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., 1991.
17. D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
18. Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.
19. D. J. Fremont, M. N. Rabe, and S. A. Seshia. Maximum model counting. In S. P. Singh and S. Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 3885–3892. AAAI Press, 2017.
20. R. M. Karp, M. Luby, and N. Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429–448, 1989.
21. V. Klebanov, N. Manthey, and C. J. Muise. SAT-based analysis and quantification of information flow in programs. In *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, pages 177–192, 2013.
22. V. Klebanov, A. Weigl, and J. Weisbarth. Sound probabilistic #SAT with projection. In *Proceedings 14th International Workshop Quantitative Aspects of Programming Languages and Systems, QAPL 2016, Eindhoven, The Netherlands, April 2-3, 2016.*, pages 15–29, 2016.
23. P. Malacaria, M. Tautchning, and D. Distefano. Information leakage analysis of complex C code and its application to openssl. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, pages 909–925, 2016.
24. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI 2008, Tucson, AZ, USA, June 7-13, 2008*, pages 193–205, 2008.
25. Z. Meng and G. Smith. Calculating bounds on information leakage using two-bit patterns. In *Proceedings of the 2011 Workshop on Programming Languages and Analysis for Security, PLAS 2011, San Jose, CA, USA, 5 June, 2011*, page 1, 2011.
26. J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 73–85, 2009.
27. Q. Phan and P. Malacaria. Abstract model counting: a novel approach for quantification of information leaks. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 283–292, 2014.
28. J. Sass. The role of static analysis in Heartbleed. `https://www.sans.org/reading-room/whitepapers/threats/role-static-analysis-heartbleed-35752`. Accessed: 2017-04-03.
29. J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas.*, pages 331–340, 1993.
30. M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th*

International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings, pages 244–257, 2009.
31. C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu. Precisely measuring quantitative information flow: 10K lines of code and beyond. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 31–46, 2016.
32. L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, 8(3):410–421, 1979.
33. J. Wang, M. Zhao, Q. Zeng, D. Wu, and P. Liu. Risk assessment of buffer "Heartbleed" over-read vulnerabilities. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015*, pages 555–562, 2015.
34. A. Weigl. Efficient SAT-based pre-image enumeration for quantitative information flow in programs. In *Data Privacy Management and Security Assurance - 11th International Workshop, DPM 2016 and 5th International Workshop, QASA 2016, Heraklion, Crete, Greece, September 26-27, 2016, Proceedings*, pages 51–58, 2016.
35. D. A. Wheeler. How to prevent the next Heartbleed. `https://www.dwheeler.com/essays/heartbleed.html`. Accessed: 2017-04-03.