Speeding Up SMT-Based Quantitative Program Analysis

Daniel J. Fremont and Sanjit A. Seshia

University of California, Berkeley
dfremont@berkeley.edu
sseshia@eecs.berkeley.edu

Abstract

Quantitative program analysis involves computing numerical quantities about individual or collections of program executions. An example of such a computation is quantitative information flow analysis, where one estimates the amount of information leaked about secret data through a program's output channels. Such information can be quantified in several ways, including channel capacity and (Shannon) entropy. In this paper, we formalize a class of quantitative analysis problems defined over a weighted control flow graph of a loop-free program. These problems can be solved using a combination of path enumeration, SMT solving, and model counting. However, existing methods can only handle very small programs, primarily because the number of execution paths can be exponential in the program size. We show how path explosion can be mitigated in some practical cases by taking advantage of special branching structure and by novel algorithm design. We demonstrate our techniques by computing the channel capacities of the timing side-channels of two programs with extremely large numbers of paths.

1 Introduction

Quantitative program analysis involves computing numerical quantities that are functions of individual or collections of program executions. Examples of such problems include computing worst-case or average-case execution time of programs, and quantitative information flow, which seeks to compute the amount of information leaked by a program. Much of the work in this area has focused on extremal quantitative analysis problems — that is, problems of finding worst-case (or best-case) bounds on quantities. However, several problems involve not just finding extremal bounds but computing functions over multiple (or all) executions of a program. One such example, in the general area of quantitative information flow, is to estimate the entropy or channel capacity of a program's output channel. These quantitative analysis problems are computationally more challenging, since the number of executions (for terminating programs) can be very large, possibly exponentially many in the program size.

In this paper, we present a formalization and satisfiability modulo theories (SMT) based solution to a family of quantitative analysis questions for deterministic, terminating programs. The formalization is covered in detail in Section 2, but we present some basic intuition here. This family of problems can be defined over a weighted graph-based model of the program. More specifically, considering the program's control flow graph, one can ascribe weights to nodes or edges of the graph capturing the quantity of interest (execution time, number of bits leaked, memory used, etc.) for basic blocks. Then, to obtain the quantitative measure for a given program path, one sums up the weights along that path. Furthermore, in order to count the number of program inputs (and thus executions) corresponding to a program path, one can perform model counting on the formula encoding the path condition. Finally, to compute the quantity of interest (such as entropy or channel capacity) for the overall program, one combines the quantities and model counts obtained for all program paths using a prescribed formula.

The obvious limitation of the basic approach sketched above is that, for programs with substantial branching structure, the number of program paths (and thus, executions) can be exponential in the program size. We address this problem in the present paper with two ideas. First, we show how a certain type of "confluent" branching structure which often occurs in real programs can be exploited to gain significant performance enhancements. A common example of this branching structure is the presence of a conditional statement inside a for-loop, which leads to 2^N paths for N loop iterations. In this case, if the branches are proved to be "independent" of each other (by invoking an SMT solver), then one can perform model counting of individual branch conditions rather than of entire path conditions, and then cheaply aggregate those model counts. Secondly, to compute a quantity such as channel capacity, it is not necessary to derive the entire distribution of values over all paths. For this case, we give an efficient algorithm to compute all the values attained by a given quantity (e.g. execution time) over all possible paths — i.e., the support of the distribution — which runs in time polynomial in the sizes of the program and the support. Our algorithmic methods are particularly tuned to the analysis of timing side-channels in programs. Specifically, we apply our ideas to computing the channel capacity of timing side-channels for two standard programs which have far too many paths for previous techniques to handle.

Our techniques enable the use of SMT methods in a new application, namely quantitative program analyses such as assessing the feasibility of side-channel attacks. While SMT methods are used in other program verification problems with exponentially-large search spaces, naïve attempts to use them to compute statistics like those we consider do not circumvent path explosion. The optimizations that form our primary contributions are essential in making feasible the application of SMT to our domain.

To summarize, the main contributions of this paper include:

- a method for utilizing special branching structure to reduce the number of model counter invocations needed to compute the distribution of a class of quantitative measures from potentially exponential to linear in the size of the program, and
- an algorithm which exploits this structure to compute the support of such distributions in time polynomial in the size of the program and the support.

The rest of the paper is organized as follows. We present background material and problem definitions in Sec. 2. Algorithms and theoretical results are presented in Sec. 3. Experimental results are given in Sec. 4 and we conclude in Sec. 5.

2 Background and Problem Definition

We present some background material in Sec. 2.1 and the formal problem definitions in Sec. 2.2.

2.1 Preliminaries

We assume throughout that we are given a loop-free deterministic program F whose input is a set of bits I. Our running example for F will be the standard algorithm for modular exponentiation by repeated squaring, denoted `modexp`, where the base and modulus are fixed and the input is the exponent. Usually `modexp` is written with a loop that iterates once for each bit of the exponent. To make `modexp` loop-free we unroll its loop, yielding for a 2-bit exponent the program shown on the left of Figure 1. Lines 2–5 and 6–9 correspond to the two iterations of the loop.

```
1: $r \leftarrow 1$
2: **if** $(e \ \& \ 1) = 1$ **then**
3:     $r \leftarrow rb \pmod m$
4: $e \leftarrow e >> 1$
5: $b \leftarrow b^2 \pmod m$
6: **if** $(e \ \& \ 1) = 1$ **then**
7:     $r \leftarrow rb \pmod m$
8: $e \leftarrow e >> 1$
9: $b \leftarrow b^2 \pmod m$
10: **return** $r$
```

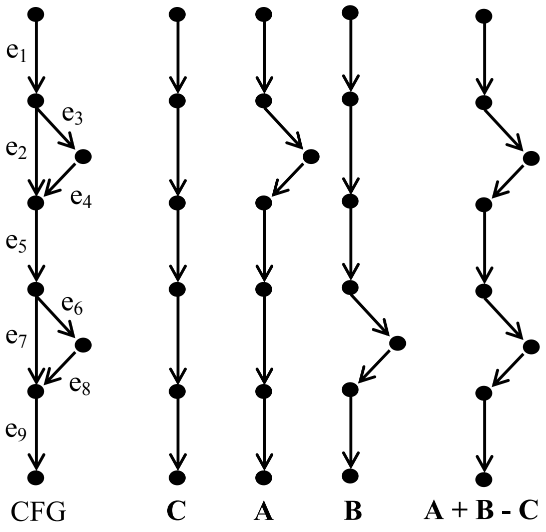

Figure 1: Unrolled pseudocode and CFG for `modexp`, computing $b^e \pmod m$ for a 2-bit exponent e. Paths **A**, **B**, and **C** form a basis, the remaining (rightmost) path being a linear combination of them.

To describe the execution paths of F we use the formalism introduced by McCabe [6]. Consider the control-flow graph (CFG) of F, where there is a vertex for each basic block, conditionals having two outgoing edges. For example, since 2-bit `modexp` has two conditionals, its CFG (shown in Figure 1) has two vertices with outdegree 2. We call such vertices *branch points*, and denote the set of them by B. Which edge out of a branch point $b \in B$ is taken depends on the truth of its *branch condition* C_b, the condition in the corresponding conditional statement. In Figure 1, the branch condition for the first branch point is $(e\&1) = 1$: if this holds, then edge e_3 is taken, and otherwise edge e_2 is taken. We model the finite-precision semantics of programs, variables being represented as bitvectors, so that the branch conditions can be expressed as bitvector SMT formulae. Since these conditions can depend on the result of prior computations (e.g. the second branch condition in Figure 1), the corresponding SMT formulae include constraints encoding how those computations proceed. Then each formula uniquely determines the truth of its branch condition given an assignment to the input bits. When necessary, these formulae can be bit-blasted into propositional SAT formulae for further analysis (e.g. model counting).

For convenience we add a dummy vertex to the CFG which has an incoming edge from all sink vertices. Since F is loop-free the CFG is a DAG, and each execution of F corresponds to a simple path from the source to the (now unique) sink. Given such a path P, we write $\mathrm{B}(P)$ for the set of branch points where P takes the right of the two outgoing edges, corresponding to making C_b true. If there are N edges then these paths can be viewed as vectors in $\{0,1\}^N$, where each coordinate specifies whether the corresponding edge is taken. For example, in Figure 1 path **A** corresponds to the vector $(1,0,1,1,1,1,0,0,1)$ under the given edge labeling. This representation allows us to speak meaningfully about linear combinations of paths, as long as the result is in $\{0,1\}^N$. A *basis* of the set of paths is defined by analogy to vector spaces to be a minimal set of paths from which all paths can be obtained by taking linear combinations. In Figure 1, the paths **A**, **B**, and **C** form a basis, as the only other path through the CFG can be expressed as $\mathbf{A} + \mathbf{B} - \mathbf{C}$.

Now suppose we are given an integer weight for each basic block of F, or equivalently for each vertex of its CFG.[1] We define the *total weight* $\mathrm{wt}(P)$ of an execution path P of F to be the sum of the weights of all basic blocks along P. Note that we get the same value if the weight of each vertex is moved to all of its outgoing edges (obviously excluding the dummy sink), and we sum edge instead of vertex weights — thus $\mathrm{wt}(\cdot)$ is a linear function. Since F is deterministic, each input $x \in \{0,1\}^I$ triggers a unique execution path we denote $\mathrm{P}(x)$, and so has a well-defined total weight $\mathrm{wt}(x) = \mathrm{wt}(\mathrm{P}(x))$.

2.2 Problem Definition

We consider in this paper the following problems:

Problem 1. Picking $x \in \{0,1\}^I$ uniformly at random, what is the distribution of $\mathrm{wt}(x)$?

and the special case:

Problem 2. What is the support of the distribution of $\mathrm{wt}(x)$, i.e. what is the set $\mathrm{wt}(\{0,1\}^I) = \{\mathrm{wt}(x) \mid x \in \{0,1\}^I\}$?

One way to think about these problems is to view the weight of a basic block as some quantity or resource, say execution time or energy, that the block consumes when executed. Then Problem 1 is to find the distribution of the total execution time or energy consumption of the program.

Computing or estimating this distribution is useful in a range of applications (see [10]). We consider here a quantitative information flow (QIF) setting, with an adversary who tries to recover x from $\mathrm{wt}(x)$. In the example above, this would be a timing side-channel attack scenario where the adversary can only observe the total execution time of the program. Given the distribution of $\mathrm{wt}(x)$, we can compute any of the standard QIF metrics such as *channel capacity* or *Shannon entropy* measuring how much information is leaked about x. For deterministic programs, the channel capacity[2] is simply the (base 2) logarithm of the number of possible observed values [11]. Thus to compute the channel capacity we do not need to know the full distribution of $\mathrm{wt}(x)$, but only how many distinct values it can take — hence our isolation of Problem 2. As we will see, this special case can sometimes be solved much more rapidly than by computing the full distribution.

We note that the general problems above can be applied to a variety of different types of resources. On platforms where the execution time of a basic block is constant (i.e. not dependent on the state of the machine), they can be applied to timing analysis. The weights could also represent the size of memory allocations, or the number of writes to a stream or device. For all of these, solving Problems 1 and 2 could be useful for performance characterization and analysis of side-channel attacks.

3 Algorithms and Theoretical Results

The simplest approach to Problem 1 would be to execute program F on every $x \in \{0,1\}^I$, computing the total weight of the triggered path and eventually obtaining the entire map

[1]Note that our formalism and approach can be made to work with rational weights, but we focus here on applications for which integer weights suffice.

[2]Sometimes called the *conditional min-entropy* of x with respect to $\mathrm{wt}(x)$, since for deterministic programs with a uniform input distribution they are the same [11].

$x \mapsto \mathrm{wt}(x)$. This is obviously impractical when there are more than a few input bits, and is wasteful because often many inputs trigger the same execution path. A more refined approach is to enumerate all execution paths, and for each path compute how many inputs trigger it. This can be done by expressing the branch conditions corresponding to the path as a bitvector or propositional formula and applying a *model counter* [3] (this idea was used in [1] to count how many inputs led to a given output, although with a linear integer arithmetic model counter). If the number of paths is much less than $2^{|I|}$, as is often the case, this approach can be significantly more efficient than brute-force input enumeration. However, as noted above the number of paths can be exponential in the size of F, in which case this approach requires exponentially-many calls to the model counter and therefore is also impractical.

A prototypical example of path explosion is our running example `modexp`. For an N-bit exponent, there are N conditionals, and all possible combinations of these branches can be taken, so that there are 2^N execution paths. This makes model counting each path infeasible, but observe that the algorithm's branching structure has two special properties. First, the conditionals are *unnested*: the two paths leading from each conditional always converge prior to the next one. Second, the branch conditions are *independent*: they depend on different bits of the input. Below we show how we can use these properties to gain greater efficiency, yielding Algorithms 2 and 4 for Problems 1 and 2 respectively.

3.1 Unnested Conditionals

If F has no nested conditionals, its CFG has an "N-diamond" form like that shown in Figure 1 (the number of basic blocks within and between the "diamonds" can vary, of course — in particular, we do not assume that the "else" branch of a conditional is empty, as is the case for `modexp`). This type of structure naturally arises when unrolling a loop with a conditional in the body, as indeed is the case for `modexp`. Verifying that there are no nested conditionals is a simple matter of traversing the CFG.

With unnested conditionals, there is a one-to-one correspondence between execution paths and subsets of B, given by $P \mapsto \mathrm{B}(P)$. For any $b \in B$, we write B_b for the path which takes the left edge at every branch point except b (i.e. makes every branch condition false except for that of b — of course it is possible that no input triggers this path). We write B_{none} for the path which always takes the left edge at each branch point. For example, in Figure 1 if the conditionals on lines 2 and 6 correspond to branch points a and b respectively, then $\mathbf{A} = B_a$, $\mathbf{B} = B_b$, and $\mathbf{C} = B_{\text{none}}$. In general, B_{none} together with the paths B_b form a basis for the set of all paths. In fact, for any path P it is easy to see that

$$P = \left(\sum_{c \in \mathrm{B}(P)} B_c \right) - \left(|\mathrm{B}(P)| - 1 \right) B_{\text{none}} \ . \qquad (1)$$

This representation of paths will be useful momentarily.

3.2 Independence

Recall that an input variable of a Boolean function is a *support variable* if the function actually depends on it, i.e. the two cofactors of the function with respect to the variable are not equivalent. For each branch point $b \in B$, let $S_b \subseteq I$ be the set of input bits which are support variables of C_b. We make the following definition:

Definition 1. Two conditionals $b, c \in B$ are *independent* if $S_b \cap S_c = \emptyset$.

Independence simply means that there are no common support variables, so that the truth of one condition can be set independently of the truth of the other.

To compute the supports of the branch conditions and check independence, the simplest method is to iterate through all the input bits, checking for each one whether the cofactors of the branch condition with respect to it are inequivalent using an SMT query in the usual way. This can be substantially streamlined by doing a simple dependency analysis of the branch condition in the source of F, to determine which input variables are involved in its computation. Then only input bits which are part of those variables need be tested (for example, in Figure 1 both branch conditions depend only on the input variable e, and if there were other input variables the bits making them up could be ignored). This procedure is outlined as Algorithm 1. Note that as indicated in Sec. 2.1, the formula ϕ computed in line 6 encodes the semantics of F so that the truth of C_b (equivalently, the satisfiability of ϕ) is uniquely determined by an assignment to the input bits. For lack of space, the proofs of Lemma 1 and the other lemmas in this section are deferred to the Appendix found in the full version of this paper.

Algorithm 1 FindConditionSupports(F)

```
 1: Compute CFG of $F$ and identify branch points $B$
 2: if there are nested conditionals then
 3:     return FAILURE
 4: for all $b \in B$ do
 5:     $S_b \leftarrow \emptyset$   // these are global variables
 6:     $\phi \leftarrow$ SMT formula representing $C_b$
 7:     $V \leftarrow$ input bits appearing in $\phi$
 8:     for all $v \in V$ do
 9:         if the cofactors of $C_b$ w.r.t. $v$ are not equivalent then
10:             $S_b \leftarrow S_b \cup \{v\}$
11: return SUCCESS
```

Lemma 1. *Algorithm 1 computes the supports S_b correctly, and given an SMT oracle runs in time polynomial in $|F|$ and $|I|$.*

If all of the conditionals of F are pairwise independent, then I can be partitioned into the pairwise disjoint sets S_b and the set of remaining bits which we write S_{none}. For any $b \in B$, the truth of C_b depends only on the variables in S_b, and we denote by T_b the number of assignments to those variables which make C_b true. Then we have the following formula for the probability of a path:

Lemma 2. *Picking $i \in \{0,1\}^I$ uniformly at random, for any path P, the probability that the path corresponding to input i is P is given by*

$$\Pr[\mathrm{P}(i) = P] = \left[2^{|S_{\text{none}}|} \left(\prod_{b \in \mathrm{B}(P)} T_b \right) \left(\prod_{b \in B \setminus \mathrm{B}(P)} \left(2^{|S_b|} - T_b \right) \right) \right] / 2^{|I|} \ .$$

Lemma 2 allows us to compute the probability of any path as a simple product if we know the quantities T_b. Each of these in turn can be computed with a single call to a model counter, as done in Algorithm 2.

Theorem 1. *Algorithm 2 correctly solves Problem 1, and given SMT and model counter oracles runs in time polynomial in $|F|$, $|I|$, and the number of execution paths of F. The model counter is only queried $|B|$ times.*

Algorithm 2 FindWeightDistribution(F, *weights*)

```
 1: if FindConditionSupports($F$) = FAILURE then
 2:     return FAILURE
 3: if the sets $S_b$ are not pairwise disjoint then
 4:     return FAILURE
 5: for all $b \in B$ do
 6:     $T_b \leftarrow$ model count of $C_b$ over the variables in $S_b$
 7: $dist \leftarrow$ constant zero function
 8: for all execution paths $P$ do
 9:     $p \leftarrow$ probability of $P$ from Lemma 2
10:     $dist \leftarrow dist[\mathrm{wt}(P) \mapsto dist(\mathrm{wt}(P)) + p]$
11: return $dist$
```

Proof. Follows from Lemmas 1 and 2. □

Algorithm 2 improves on path enumeration by using one invocation of the model counter per branch point, instead of one invocation per path. In total the algorithm may still take exponential time, since we need to compute the product of Lemma 2 for each path, but if model counting is expensive there is a substantial savings.

Further savings are possible if we restrict ourselves to Problem 2. For this, we want to compute the possible values of $\mathrm{wt}(x)$ for all inputs x. This is identical to the set of possible values $\mathrm{wt}(P)$ for all *feasible* paths P (the paths that are executed by some input). Thus, we do not need to know the probability associated with each individual path, but only which paths are feasible and which are not. Lemma 2 implies that all paths are feasible (unless some $T_b = 0$ or $T_b = 2^{|S_b|}$, corresponding to a conditional which is identically false or true; then $S_b = \emptyset$, so we can detect and eliminate such trivial conditionals), and this leads to

Lemma 3. *Let D be the multiset of differences $\mathrm{wt}(B_b) - \mathrm{wt}(B_{\mathrm{none}})$ for $b \in B$. Then the possible values of $\mathrm{wt}(i)$ over all inputs $i \in \{0,1\}^I$ are the possible values of $\mathrm{wt}(B_{\mathrm{none}}) + D^+$, where D^+ is the set of sums of submultisets of D.*

To use Lemma 3 to solve Problem 2, we must find the set D^+. The brute-force approach of enumerating all submultisets is obviously impractical unless D is very small. We cannot hope to do better than exponential time in the worst case[3], since D^+ can be exponentially larger than D. However, in many practical situations D^+ is not too much larger than D. This is because the paths B_b often have similar weights, so the variation $V = \max D - \min D$ is small and we can apply the following lemma:

Lemma 4. *If $V = \max D - \min D$, then $|D^+| = O(V\,|D|^2)$.*

Small differences between weights are exploited by Algorithm 3, which as shown in the Appendix computes D^+ in $O(|D|\,|D^+|)$ time. By Lemma 4, the algorithm's runtime is $O(|D| \cdot V|D|^2) = O(V\,|D|^3)$, so it is very efficient when V is small. The essential idea of the algorithm is to handle one element $x \in D$ at a time, keeping a list of possible sums found so far sorted so that updating it with the new sums possible using x is a linear-time operation. For simplicity we only show how positive $x \in D$ are handled, but see the analysis in the Appendix for the general case.

[3]Although we note that for channel capacity analysis we only need $|D^+|$ and not D^+ itself, and there could be a faster (potentially even polynomial-time) algorithm to find this value.

Algorithm 3 SubmultisetSums(D)

```
1:  $sums \leftarrow (0)$
2:  for all $x \in D$ do
3:      $newSums \leftarrow (sums[0])$
4:      $i \rightarrow 1$   // index of next element of $sums$ to add to $newSums$
5:      for all $y \in sums$ do
6:          $z \leftarrow x + y$
7:          while $i < \mathsf{len}(sums)$ and $sums[i] < z$ do
8:              $newSums.\mathsf{append}(sums[i])$
9:              $i \leftarrow i + 1$
10:         $newSums.\mathsf{append}(z)$
11:         if $i < \mathsf{len}(sums)$ and $sums[i] = z$ then
12:             $i \leftarrow i + 1$
13:     $sums \leftarrow newSums$
14: return $sums$
```

Using Algorithm 3 together with Lemma 3 gives an efficient algorithm to solve Problem 2, outlined as Algorithm 4. This algorithm has runtime polynomial in the size of its input and output.

Algorithm 4 FindPossibleWeights($F, weights$)

```
1:  if FindConditionSupports($F$) = FAILURE then
2:      return FAILURE
3:  if the sets $S_b$ are not pairwise disjoint then
4:      return FAILURE
5:  Eliminate branch points with $S_b = \emptyset$ (trivial conditionals)
6:  $D \leftarrow$ empty multiset
7:  for all $b \in B$ do
8:      $d \leftarrow \mathrm{wt}(B_b) - \mathrm{wt}(B_{\mathrm{none}})$
9:      $D \leftarrow D \cup \{d\}$
10: $D^+ \leftarrow$ SubmultisetSums($D$)
11: return $\mathrm{wt}(B_{\mathrm{none}}) + D^+$
```

Theorem 2. *Algorithm 4 solves Problem 2 correctly, and given an SMT oracle runs in time polynomial in $|F|$, $|I|$, and $\left|\mathrm{wt}(\{0,1\}^I)\right|$.*

Proof. Clear from Lemmas 1 and 3, and the analysis of Algorithm 3 (see the Appendix). □

3.3 More General Program Structure

As presented above, our algorithms are restricted to loop-free programs which have only unnested, independent conditionals. However, our techniques are still helpful in analyzing a large class of more general programs. Loops with a bounded number of iterations can be unrolled. Unrolling the common program structure consisting of a for-loop with a conditional in the body yields a loop-free program with unnested conditionals. If the conditionals are pairwise independent, as in the `modexp` example, our methods can be directly applied. If the number of dependent conditionals, say D, is nonzero but relatively small, then each of the 2^D assignments

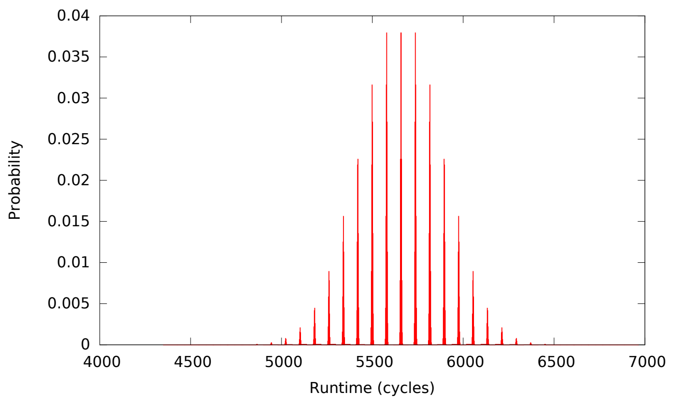

Figure 2: Timing distribution of 32-bit `modexp`, computed with Algorithm 2.

to these conditionals can be checked for feasibility with an SMT query, and the remaining conditionals can be handled using our algorithms. If many conditionals are dependent then checking all possibilities requires an exponential amount of work, but we can efficiently handle a limited failure of independence. An example where this is the case is the Mersenne Twister example we discuss in Sec. 4, where 2 out of 624 conditionals are dependent. A small level of conditional nesting can be handled in a similar way. In general, when analyzing a program with complex branching structure, our methods can be applied to those regions of the program which satisfy our requirements. Such regions do frequently occur in real-world programs, and thus our techniques are useful in practice.

4 Experiments

As mentioned in Sec. 2, Problem 2 subsumes the computation of the channel capacity of the timing side-channel on a platform where basic blocks have constant runtimes. To demonstrate the effectiveness of our techniques, we use them to compute the timing channel capacities of two real-world programs on the *PTARM* simulator [4]. The tool *GameTime* [9] was used to generate SMT formulae representing the programs, and to interface with the simulator to perform the timing measurements of the basis paths. SMT formulae for testing cofactor equivalence were generated and solved using *Z3* [2]. Model counting was done by using Z3 to convert SMT queries to propositional formulae, which were then given to the model counter *Cachet* [8]. Raw data from our experiments can be obtained at `http://math.berkeley.edu/~dfremont/SMT2014Data/`.

The first program tested was the `modexp` program already described above, using a 32-bit exponent. With 2^{32} paths, enumerating and model counting all paths is clearly infeasible. Our new approach was quite fast: finding the branch supports, model counting[4], and running Algorithm 3 took only a few seconds, yielding a timing channel capacity of just over 8 bits. In fact, although the number of paths is very large, the per-path cost of Algorithm 2 is so low that we were able to compute `modexp`'s entire timing distribution with it in 23 hours (effectively analyzing more than 50,000 paths per second). The distribution is shown in Figure 2.

[4]We note that for this program, each branch condition had only a single support variable, and thus we have $T_b = 1$ automatically without needing to do model counting.

The second program we tested was the state update function of the widely-used pseudorandom number generator the Mersenne Twister [5]. We tested an implementation of the most common variant, `MT19937`, which is available at [7]. On every 624th query to the generator, `MT19937` performs a nontrivial updating of its internal state, an array of 624 32-bit integers. We analyzed the program to see how much information about this state is leaked by the time needed to do the update. The relevant portion of the code has 2^{624} paths and thus would be completely impossible to analyze using path enumeration. With our techniques the analysis became feasible: finding the branch supports took 54 minutes, while Algorithm 3 took only 0.2 seconds because there was a high level of uniformity across the path timings. The channel capacity was computed to be around 9.3 bits. We note that among the 624 branch conditions there are two which are not independent. Thus all four truth assignments to these conditions needed to be checked for feasibility before applying our techniques to the remaining 622 conditionals.

5 Conclusions

We presented a formalization of certain quantitative program analysis problems that are defined over a weighted control-flow graph representation. These problems are concerned with understanding how a quantitative property of a program is distributed over the space of program paths, and computing metrics over this distribution. These computations rely on the ability to solve a set of satisfiability (SAT/SMT) and model counting problems. Previous work along these lines has only been applicable to small programs with very few conditionals, since it typically depends on enumerating all execution paths and the number of these can be exponential in the size of the program. We investigated how in certain situations where the number of paths is indeed exponential, special branching structure can be exploited to gain efficiency. When the conditionals are unnested and independent, we showed how the number of expensive model counting calls can be reduced to be linear in the size of the program, leaving only a very fast product computation to be done for each path. Furthermore, a special case of the general problem, which for example is sufficient for the computation of side-channel capacities, can be solved avoiding exponential path enumeration entirely. Finally, we showed the practicality of our methods by using them to compute the timing side-channel capacities of two commonly-used programs with very large numbers of paths.

Acknowledgements

Daniel wishes to thank Jon Kotker, Rohit Sinha, and Zach Wasson for providing assistance with some of the tools used in our experiments, and Garvit Juniwal for a useful discussion of Algorithm 3 and Lemma 4. The authors also thank the anonymous reviewers for their helpful comments and suggestions. This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy*, pages 141–153. IEEE, 2009.

[2] L. de Moura and N. Bjørner. Z3. `http://z3.codeplex.com/`.

[3] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. *Handbook of Satisfiability*, pages 633–654, 2009.

[4] E. A. Lee and S. A. Edwards. PTARM simulator. `http://chess.eecs.berkeley.edu/pret/src/ptarm-1.0/ptarm_simulator.html`.

[5] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[6] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[7] T. Nishimura and M. Matsumoto. Implementation of MT19937. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/MTARCOK/mt19937ar-cok.c`.

[8] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[9] S. A. Seshia and J. Kotker. GameTime: A toolkit for timing analysis of software. In *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.

[10] S. A. Seshia and A. Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(S2):55:1–55:27, 2012.

[11] G. Smith. On the foundations of quantitative information flow. In *Foundations of Software Science and Computational Structures*, pages 288–302. Springer, 2009.