CALCULATING BOUNDS ON INFORMATION LEAKAGE USING MODEL-CKECINGG TOOLS

A Thesis presented to
the Faculty of the Graduate School
at the University of Missouri

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

---

by

JIA CHEN

Dr. Rohit Chadha, Thesis Supervisor

JUL 2014

The undersigned, appointed by the Dean of the Graduate School, have examined the thesis entitled:

CALCULATING BOUNDS ON INFORMATION LEAKAGE
USING MODEL-CKECINGG TOOLS

presented by Jia Chen,

a candidate for the degree of Master of Science and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Rohit Chadha

Dr. Prasad Calyam

Dr. Michela Becchi

ACKNOWLEDGMENTS

This page is where you would acknowledge all those who helped you with your academic research. This is not necessarily where you would recognize loved ones who supported you during your studies. That would be more appropriately done in an optional Dedication page. I would like to thank Professor Smith Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Vestibulum eu tellus. Nullam et odio eget sapien porttitor interdum. Donec vel ante. Maecenas in sem a nunc viverra hendrerit. Quisque ut massa quis pede blandit pharetra.

Pellentesque sed ligula sit amet ligula scelerisque sagittis. Nulla adipiscing tellus at pede. Cras id nunc vel diam congue dictum. Donec a nulla nec eros ornare consequat. Nullam quis orci. Nam adipiscing, erat in congue pellentesque, dolor eros euismod quam, a egestas mauris magna varius justo.

Sed eu sem et lorem blandit volutpat. Duis pulvinar, arcu quis suscipit convallis, ante elit auctor dui, in fermentum diam velit a mauris. In risus odio, consectetuer quis, ullamcorper in, rutrum ut, metus.

TABLE OF CONTENTS

ACKNOWLEDGMENTS . . . . **ii**

LIST OF TABLES . . . . **v**

LIST OF FIGURES . . . . **vi**

ABSTRACT . . . . **vii**

CHAPTER

1 Introduction **1**

2 Theory background **2**

2.1 Min-entropy 2

2.2 Information leak 2

2.3 Two intuitive solutions 2

2.3.1 Double loop and counter 3

2.3.2 Single loop and table 4

3 Experiment with Getafix **6**

3.1 Getafix 7

3.2 The converter 7

3.3 Tests and results 10

3.3.1 Sanity check 10

4 Monotonic programs **11**

5 Summary and concluding remarks **12**

APPENDIX

A Title of first appendix **13**

A.1 Section title 13

A.1.1 Subsection title 13

BIBLIOGRAPHY **14**

VITA **15**

LIST OF TABLES

Table Page

3.1 Built-in function names and corresponding operators 10

LIST OF FIGURES

Figure Page

ABSTRACT

This is the abstract of your dissertation project. It should not exceed one page.

Chapter 1

Introduction

Introduce the reader to the current problem that you wish to solve, and why anyone should care about it.

Chapter 2

Theory background

2.1 Min-entropy

The introduction counts as chapter 1. This page shows how the bulk of your thesis will be organized: through chapters and sections. Here is a citation.[**?**]

2.2 Information leak

2.3 Two intuitive solutions

To count the number of outputs of a program, we come up with two approaches: Put the program in a double loop and count the number of outputs, or iterate through all input values and record the outputs in a table. The first approach is time-consuming, while the second one is space-consuming.

2.3.1 Double loop and counter

In algorithm 1, for each possible output value, we iterate through the input range to see if an input can result in this output. If we hit such an input, counter increases and the code breaks out of the inner loop to continue testing the next possible output value. After the double loop finishes, the value of $OCounter$ is the number of outputs of program P.

Algorithm 1 Calculate the number of outputs using double loop.

```
S ← 0
O ← 0
SIn ← 0
OOut ← 0
OCounter ← 0
SMax ← 1 << bitLength − 1
OMax ← 1 << bitLength − 1
for O = 0 to OMax do
  for S = 0 to SMax do
    SIn ← S
    OOut ← P(SIn) // the program P takes SIn as input
    if OOut = O then
      OCounter ← OCounter + 1
      break
    end if
  end for
end for
```

In algorithm 1, we declared seven variables, and all of them requires $bitLength$ bits except for $OCounter$ which is $bitLength + 1$ bits. The total memory usage for variables is $7 \times bitLength + 1$ at $O(bitLength)$. As with execution time, we assume program P takes time $t(P)$ to execute, and the total execution time for the double loop when break is never reached is $2^{bitLength} \times 2^{bitLength} \times t(P)$. Thus the time complexity is $(2^{O(bitLength)}) \times t(P)$.

In order to get an estimation of how much time the double loop will take to execute, we implemented a piece of C code with an empty while loop which loops 2^{32} times. On out experiment PC, this loop takes on average 10.30 seconds to complete. Were we to run a double loop in bit length of 32, the execution time would be $2^{32} \times 10.30$ seconds, which is around 1403 years. Running the double loop directly would be infeasible.

2.3.2 Single loop and table

In algorithm 2, we create a table with size equal to the maximum number of possible outputs($1 << bitLength$, or $2^{bitLength}$), and we use its indices as output values. $OHit[O] = 1$ means O is an output for program P. When a 0 turns to 1, we increase $OCounter$. After the loop, the value of $OCounter$ is the number of outputs by program P.

Algorithm 2 Calculate the number of outputs using single loop and a table.

$S \leftarrow 0$
$O \leftarrow 0$
$SIn \leftarrow 0$
$OOut \leftarrow 0$
$OCounter \leftarrow 0$
$SMax \leftarrow 1 << bitLength - 1$
$OMax \leftarrow 1 << bitLength - 1$
$OHit[OMax + 1] \leftarrow [0]$
for $S = 0$ to $SMax$ **do**
  $SIn \leftarrow S$
  $OOut \leftarrow P(SIn)$ // the program P takes SIn as input
  if $OHit[OOut] = 0$ **then**
    $OCounter \leftarrow OCounter + 1$
    $OHit[OOut] \leftarrow 1$
  end if
end for

In algorithm 2 except for the array we have 7 variables using $7 \times bitLength + 1$ memory. The array $OHit[]$ is of size $2^{bitLength} \times bitLength$ making a total of $2^{bitLength} \times bitLength + 7 \times bitLength + 1$ at $2^{O(bitLength)}$. As with execution time, we assume program P takes time $t(P)$ to execute, and the execution time for the single loop is $2^{bitLength} \times t(P)$. Thus the time complexity is $(2^{O(bitLength)}) \times t(P)$.

We set the bit length to 32. In array $OHit[]$, each element is 32 bits and the number of elements is 2^{32}. The total memory usage for this array is $2^{32} \times 32$ bits, which is 16 gigabytes. To our knowledge, it is neither difficult nor expensive to build a PC with more than 16 gigabytes of memory, and we can get such a PC off-the-shelf from top gaming PC brands like Alienware.

if this is not hard we need reasons not doing this..

Chapter 3

Experiment with Getafix

We want to use model-checking tools to see if we can reduce the time requirement of algorithm 1. Specifically, we choose the reachability property and append algorithm 3 to the end of algorithm 1. The statement within the if statement has a label. Although the exact statement following that label is irrelevant, reaching this line means $OCounter$ satisfies the constrains in the condition block.

(why and what other property are there)

We experimented on several model-checking tools, including Interproc from [1], Berkeley Lazy Abstraction Software Verification Tool(Blast) from [2] and Getafix from [3]. We can not get correct reachability results from Interproc and Blast, so we shift our focus on Getafix.

correct citations of web page

Algorithm 3 Determine if $OCounter$ meets certain constrains.

```
if value of OCounter meets certain constrains then
  reach: OCounter // a label followed by a statement
end if
```

3.1 Getafix

Getafix is a symbolic model checker for Boolean programs implemented in [3]. Getafix only supports reachability check. It translates sequential and concurrent Boolean programs into Boolean formulae and uses the model-checker Mucke to solve the reachability problem symbolically using Boolean Decision Diagrams [4].

Mostly copied from getafix website. Need rewrite

3.2 The converter

Input for Getafix are boolean programs, meaning it only supports binary variables which can be either 0 or 1. We represent our problem in c-style code, thus we need to translate it into boolean form. We implemented a converter to automate this process. The converter has three components, a parser, a built-in function generator and a piece of script which calls the first two components and assemble the output file.

Input to the parser is the c-style code file and the desired bit length. Output of the parser is its corresponding boolean program which follows the syntax of Getafix input file. First we define the syntax of input code to the parser and second we create the parser using flex and bison. The parser scans the input code and builds a syntax tree. Then the parser prints the syntax tree as a boolean program. The parser has three points worth noting:

1. When printing the output code, the parser "stretches" each variable and literal into its binary form. Assume the desired bit length is $bitLength$. We split each variable into $bitLength$ variables by copying the name of the variable $bitLength$ times and append a counter value to each one. Also we convert a literal to

its corresponding binary value and prepend it with zeros to reach the desired length.

2. In a boolean program, all operators operate on bit level, so we need to implement higher-level operators like plus, minus, greater than and left shift using operators that Getafix supports. In the parser, we print these high-level operators as function calls in the output boolean program, and the built-in function generator generates the body of the function.

3. In the boolean code syntax which Getafix defines, function call plus the semi-colon is defined as a statement, and another rule allows the code to assign a function call to an identifier, but function call itself is not an expression. This means that a function call can not work as an expression as in many other languages, and it leads to two problems: First, the decider expression in if...else and while statements can not contain function calls. Second, parameters of a function call or operands to an operator can not be a function call. We automated a solution in the parser to the first problem, which assigns the decider expression to a temporary variable and use that variable as the decider, so we can use the c-style if...else and while in the input code. For the second problem, a possible solution would be to manage a set of internal temporary variables and assign each function call to a variable, but we did not implement it.

Input to the built-in function generator is the desired bit length. Output is a set of high-level operators like plus and left shift implemented as functions. We do not track the necessary functions in the parser, as experiments with Getafix indicate that the uncalled functions affect little on the execution time. Listing 3.1 shows a sample function by the generator, and table 3.1 shows all the operators supported by the converter.

Listing 3.1: Greater than operator as a function in boolean program with bit length of 2.

```
bool isGT(left2,left1,left0,right2,right1,right0)
begin
if (left2 != right2) then
        if (left2 = 1) then
                return 1;
        fi
else
        if (left1 != right1) then
                if (left1 = 1) then
                        return 1;
                fi
        else
                if (left0 != right0) then
                        if (left0 = 1) then
                                return 1;
                        fi
                fi
        fi
fi
return 0;
```

end

Table 3.1: Built-in function names and corresponding operators

Function name	Operator in input code
plus	+
minus	-
and	&
or	—
xor	ˆ
isGT	>
isNotEqual	!=
isEqual	==
isGTEQ	>=
isLT	<
isLTEQ	<=
lShift	<<
rShift	>>

3.3 Tests and results

3.3.1 Sanity check

Chapter 4

Monotonic programs

Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows. Some paragraph text follows.

Chapter 5

Summary and concluding remarks

Congratulations on completing your dissertation.

Appendix A

Title of first appendix

A.1 Section title

Here is some additional information which would have detracted from the point being made in the main article.

A.1.1 Subsection title

This section even has subtitles

Bibliography

[1] Interproc, 2011.

[2] MTC (models and theory of computation): BLAST project, 2008.

[3] Salvatore La Torre, Madhusudan Parthasarathy, and Gennaro Parlato. Analyzing recursive programs using a fixed-point calculus. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 211222, New York, NY, USA, 2009. ACM.

[4] Getafix – boolean program checker, 2009.

VITA

This is a summary of your *professional* life, and should be written appropriately. This can be written in the following order: where your where born, what undergraduate university you graduated from, if you received a masters, and which institution you graduated from with your PhD (University of Missouri). You can describe when you began research with your current advisor.

In another paragraph, you could say if/when you were married, what the name of your kids are, and what your plans are for after graduation if you choose. Take a look at other vita's from other dissertations for examples.