CALCULATING INFORMATION LEAKAGE
USING MODEL CHECKING 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 INFORMATION LEAKAGE
USING MODEL CHECKING 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

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A *confidential* program should not allow *any* information about its secret inputs to be inferred from its public outputs. As such confidentiality is difficult to achieve in practice, it has been proposed in literature to evaluate security of programs by computing the *amount* of information leaked. We consider the problem of *computing* information leaked by a deterministic program when the information-theoretic measure of *min-entropy* is used to quantify the amount of information.

The main challenge in computing the amount of information leaked by a program P using min-entropy is that one has to count the number of distinct possible outputs that may be observed when the program is run on different inputs. There is a polynomial-time reduction from the problem of checking whether information leaked by a program is equal to (or less than) a given number to the problem of checking safety in programs. Thus, it has been hypothesized that leakage can be estimated using model-checking tools which were originally developed for checking safety in programs.

We tested the above hypothesis using two popular model-checking tools, JMoped and Getafix. Our tests indicate that these do not scale as the number of bits in the input increases. However, we show that if the program P enjoys the additional property of *monotonicity* then we can use a different reduction to the problem of checking safety in programs. Note that if the program P inputs n bits and outputs m bits then the program P can be considered as a function P_{func} from n-bit binary numbers to m-bit binary numbers. We say P enjoys *monotonicity* if P_{func} is monotonic. We observe a dramatic improvement in the performance with this new reduction.

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 let the program iterate through all input values and record the outputs in an array. 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, $OCounter$ 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.

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

In Algorithm 1, we declare seven variables, and all of them require $bitLength$ bits except for $OCounter$ which needs to be $bitLength + 1$ bits. The total memory usage for variables is $7 \times bitLength + 1$ at $O(bitLength)$. As with execution time, we assume program P takes $t(P)$ seconds 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 $(2^{O(bitLength)}) \times t(P)$.

Bit length	Time(s)	Multiplier
14	0.708	N/A
15	2.797	3.951
16	11.196	4.003
17	44.515	3.976
18	178.970	4.020

Table 2.1: Execution time of an empty double loop with different bit length.

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 at 32 bits would be infeasible.

Starting with bit length n, the time requirement for a full double loop is $2^{2 \times n} \times t(P)$. Increase the bit length by one and the time becomes $4 \times 2^{2 \times n} \times t(P)$, four times the previous time. Table 2.1 shows the actual execution time of an empty double loop under different bit length, and the time increase follows the theoretical analysis. At bit length of 23, the execution time would exceed a day, and executing at higher bit length is impractical.

2.3.2 Single loop and array

In Algorithm 2, we create a bit array with size equal to the maximum number of possible outputs($1 << bitLength$, or $2^{bitLength}$) and initialize it with zeros. While we iterate through the range of S, we set each $OHit[P(SIn)]$ to 1. 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**
$\quad SIn \leftarrow S$
$\quad OOut \leftarrow P(SIn)$ // the program P takes SIn as input
$\quad$ **if** $OHit[OOut] = 0$ **then**
$\quad\quad OCounter \leftarrow OCounter + 1$
$\quad\quad OHit[OOut] \leftarrow 1$
$\quad$ **end if**
end for

In Algorithm 2 except for the array we have 7 variables using $7 \times bitLength + 1$ bits memory. The array $OHit[]$ is of size $2^{bitLength} \times 1$ bits making a total of $2^{bitLength} + 7 \times bitLength + 1$ bits at $2^{O(bitLength)}$. As with execution time, we assume program P takes $t(P)$ seconds 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 1 bit and the number of elements is 2^{32}. The total memory usage for this array is $2^{32} \times 1$ bits, which is 0.5 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. However, as this memory requirement grows exponentially, adding five or six bits to the bit length and the requirement will exceed the capacity of current PCs.

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

5

Chapter 3

Experiment with Getafix and jMoped

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

correct citations

Algorithm 3: Determine if $OCounter$ meets certain constrains.

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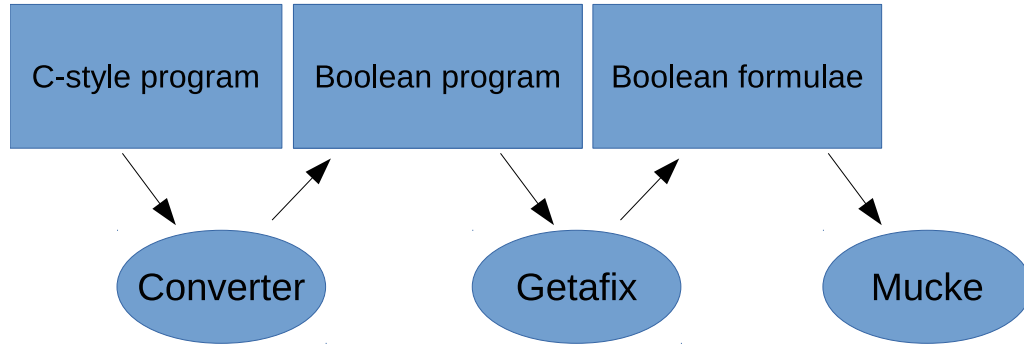   **end if**

Figure 3.1: Workflow of calculating information leakage with Getafix.

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].

3.1.1 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, and Figure 3.1 shows the workflow we used to calculate information leakage using Getafix. The converter has three components, a parser, a built-in function generator and a piece of script which calls the first two components and assembles the output file. The converter has these properties:

1. The input to the converter is a c-style code file and a positive integer which represents the bit length. The converter supports 32 bits and less. Also note

that the converter represents a number with bit length of $bitLength$ using $bitLength + 1$ bits. We do this so that we do not need to deal with the upper bound explicitly when writing a loop iterating from 0 to $1 << bitLength - 1$.

2. The output of the converter is a boolean program which follows the syntax of Getafix input file.

3. In the language that we defined for the input file, we support only one variable type: non-negative decimal integer. Again we support the length up to 32 bits.

4. Our converter does not support function definitions. The input file has two parts, variable declarations block and statements block. The parser will print these blocks into the main function of the output boolean program. Also we require all variable declarations to appear before statements in the input code. Getafix input syntax has this requirement, and we decide to keep it in our converter.

5. We implement support for the following symbolic operators in the language: plus +, minus -, and &, or |, xor ˆ, greater than >, equal ==, less than <, not equal !=, greater than or equal >=, less than or equal <=, left shift <<and right shift >>.

6. We implement support for three control statements: if...else, while loop, goto and statements with labels. We currently do not support for loop, do...while loop, switch statements, break and continue, but these statements can be easily expressed using the supported ones.

Input to the parser is a c-style code file and a 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 the input code 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:

Table 3.1: Examples of input and output of the parser, with bit length of 4

Input	Output
var SMax = 16;	decl SMax4,SMax3,SMax2,SMax1,SMax0; Smax4, SMax3, SMax2, SMax1, SMax1 := 1,0,0,0,0;
STemp = STemp - 5;	STemp4,STemp3,STemp2,STemp1,STemp0 := minus(STemp4,STemp3,STemp2,STemp1,STemp0,0,0,1,0,1);

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 appending 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. Table 3.1 contains an example of how the parser deals with a variable declaration.

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. Table 3.1 also contains an example of how the parser deals with a high-level operator.

9

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 expressions 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 a 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.

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
```

Input to the third component, a piece of script, is a c-style code file and a desired bit length. Output of the script is a boolean program ready for Getafix to process. The script first passes the bit length to the built-in function generator and redirects its output to the output file. Then the script passes both the input to the parser and appends its output to the output file. At this point the output is complete.

3.2   jMoped

jMoped is a model checker which checks for coverage in Java programs. The authors implemented it as a plug-in for eclipse, using its UI for parameters and output display. We wrap our tests in Java classes so that we can use jMoped on them.

need more explanation

3.3   Tests and results

Before we have the converter, we coded a test case manually and Getafix gave us the correct answer. Now with the converter we can run more complicated tests and see

if Getafix is a good solution to the problem of calculating information leakage. We decide to run the ten test cases from paper [?]. In each test, O represents the output of its test, and S is the input.

3.3.1 Sanity check

Listing 3.2 shows the code we use. In this test, O remains constant unless S is within a certain range. The program has 16 different outputs, ranging from 4 to 19.

Listing 3.2: Sanity check test program.

```
var base = 4;
if(S < 16){O = base+ S;}
else{O = base;}
```

Listing 3.3: Implict flow test program.

```
O = 0;
if(S == 0){O = 0;}
else{if(S == 1){O = 1;}
        else{if(S == 2){O = 2;}
                else{if(S == 3){O = 3;}
                        else{if(S == 4){O = 4;}
                                else{if(S == 5){O = 5;}
                                        else{if(S == 6){O = 6;}
}          }          }          }          }          }
```

Chapter 4

Monotonic programs

In this chapter we describe our discovery that if the test program P meets a certain property, then we can execute the program in a loop and get a direct output count in a reasonable time.

Need to insert somewhere else.

Definition 1. *Let bitLength $\in \{0, 1, 2, ..., 32\}$ and let P be a function whose input $S \in \{0, 1, 2, ..., 1 \ll bitLength - 1\}$ and output $O \in \{0, 1, 2, ..., 1 \ll bitLength - 1\}$.*

4.1 Discovery by mistake

During the initial tests with the sanity check program of bit length 32, the program terminates in less than a minute and gives the correct output count. Later we time the execution of an empty loop from 0 to $2^{32} - 1$ and the result is 10.30 seconds, so theoretically the double loop would take a much longer time to terminate, longer than our test result with sanity check. Listing 4.1 shows the C code we used.

Listing 4.1: Initial implementation of the double loop with sanity check in C.

```c
uint32_t S = 0;
uint32_t O = 0;
uint32_t SMax = S - 1;
uint32_t OMax = O - 1;
uint32_t OCounter = 0;
uint32_t OTemp = 0;
uint32_t base = 0;
for (;O<OMax;O++){
        for (;S<SMax;S++){
                //program start here
                if (S < 16)
                        OTemp = base + S;
                else
                        OTemp = base;
                //program end here
                if (OTemp == O){
                        OCounter ++;
                        break;
}       }       }
```

Due to the large difference in actual execution time and the theoretical time, we decided to inspect the code for errors. Our code differs from Algorithm 1 at two places: First, S and O can not reach $SMax$ and $OMax$ due to the use of $<$, thus the upper bound is not tested. Second, in the inner loop, we did not initialize S

14

to 0 except for its first iteration. The first point does not affect the output count of the sanity check example as every S greater than 16 will lead to a $P(S)$ value of *base* which is 0. The second point is the cause of the large time difference, because essentially P only executes once for each S, reaching a total of 2^{32} times, much smaller than the full double loop in which P executes $2^{2\times32}$ times.

We found that removing the initialization of S to 0 can be a way to speed up some test cases, but not all. Specifically, program P has to obey certain properties:

1. Function P is monotonically increasing within range $[0, n]$.

2. The output of function P with S in range $[0, n]$ has to start with zero and is continuous.

3. Each output of function P with S in range $[n + 1, 1 \ll bitLength - 1]$ can be produced by at least one S in range $[0, n]$.

4.2   Improvements

The properties that P has to follow in order for the optimization to work is rather strict. For example in the sanity check test, the optimization only works when *base* is 0. A possible solution would be to start O at the lowest value P can output, in essence pulling the function down on y axis so that it starts with 0. However, this approach does not ease the restrictions.

In our second attempt to improve this optimization method as shown in Listing 4.2, we started with monotonicity and rearranged the architecture of the outer program. The problem is to count the number of outputs of a monotonically increasing

15

function, and our solution is to iterate through the entire range of S and keep track of the largest output value. If P generates a value $OOut$ which is larger than the current largest value, then $OOut$ replaces that value and $OCounter$ increases. The properties that P has to follow in order to use this optimization is:

1. Function P is monotonically increasing within range $[n_1, n_2]$.

2. Each output of function P with S in range $[n_2 + 1, 1 \ll bitLength - 1]$ can be produced by at least one S in range $[n_1, n_2]$.

So with the second attempt, we removed the requirement for P to be continuous and start with 0 from the first attempt.

Listing 4.2: Implementation of the optimization with sanity check in C.

```
unsigned int S = 0;
unsigned int SMax = 4294967295;
unsigned int base =4;
unsigned int STemp = S;
unsigned int OTemp = 0;
//program starts here
if (STemp< 16){OTemp = base+ STemp;}
else{OTemp = base;}
//program ends here
unsigned int OCounter = 1;
unsigned int OMax = OTemp;
S++;
while (S<= SMax){
```

```
        STemp = S ;
        //program starts here
        if (STemp < 16){OTemp = base+ STemp ;          }
        else {OTemp =  base ;}
        //program ends here
        if  (OTemp > OMax ){
                OMax = OTemp;
                OCounter= OCounter+1;
        }
        if (S < SMax)  S=S+1;
        else break ;
}
```

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.