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

JULY 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

I would like to express my sincere gratitude to my advisor Dr.Rohit Chadha, for guiding me through the research process when I was working on the previous thesis subject of browser cross-site scripting defence, and furthermore for providing me the current subject when I was struggling with the previous one. He helped me all the way during research and writing of this thesis, with his deep knowledge on information security.

I also sincerely give my appreciation to Dr.Prasad Calyam and Dr.Michela Becchi, for their time and work serving as my thesis committee members.

I'd like to thank Umang Mathur for providing MOPED source code and building instructions during the tests. His email saved us plenty of time.

Last but not the least, I would like to thank my parents and friends, for encouraging me to fight on.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

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 it leaks. In this thesis, we consider the problem of computing information leaked by a deterministic program and use the information-theoretic measure of min-entropy to quantify the amount of information.

The main challenge in computing information leakage by a program using min-entropy is that one has to count the number of distinct outputs by that program. We find a polynomial-time reduction from the problem of counting outputs to the problem of checking safety in programs. Thus we propose a hypothesis that we can estimate leakage using model checking tools which are originally developed for checking safety.

We test the above hypothesis using two popular model checking tools, JMoped and Getafix. Our tests indicate that they do not scale as the number of bits in the input increases. However, we find that if the program enjoys the additional property of monotonicity then we can use a different reduction to the problem of checking safety. We observe a dramatic improvement in performance with this new reduction.

Chapter 1

Introduction

A desirable property for a program is *non-interference* [3, 6] which informally says that a program should never leak any information about its secret inputs. Formally, *non-interference* [3, 6] says that that the *low-security observations* of the executions of a program must be independent of secret inputs. However, the desired functionality of a program usually makes non-interference unachievable. For example, a password checker behaves differently on a correct password and an incorrect password (and its behavior even depends on the number of incorrect passwords entered). Therefore, many authors [2, 4, 5, 7] have proposed to evaluate security of programs by quantifying the amount of information leaked. How do we measure the amount of information leaked? How to compute the information leaked?

For measuring the amount of information leaked by programs, information-theoretic measures are often used. In this approach, a program is modeled as an information channel that transforms a random variable taking values from the set of confidential inputs into a random variable taking values from the set of public outputs. Then

information-theoretic measures are used to quantify the adversary's initial certainty about the secret inputs and the uncertainty remaining in the secret inputs after the adversary observes the execution. The amount of information leaked by the program is the difference between the two. While, many information-theoretic measures can be used, it has been argued that leakage based on min-entropy is appropriate to security applications [7]. Intuitively, leakage based on min-entropy measures vulnerability of the secret inputs to a *single* guess of the adversary who observes the program execution.

Even though quantifying information leaked in programs is appealing, it is however not easy to compute information leaked in programs. Indeed it is known that the decision problem of checking whether information leaked in non-recursive boolean programs is equal to (or less than) a given rational number is PSPACE-complete [9, 10, 1, 8]. Recall that PSPACE is the class of decision problems that can be solved by Turing machines that accesses at most a polynomial number of cells on its working tape and that this class includes NP decision problems.

In order to measure the information leaked by a program P using min-entropy, one has to count the number of different possible outputs that may be achieved when the program is run with different inputs. The amount of information leaked is the binary logarithm of this number, Now, if the input to the program P consists of n-bits, this quantity can be computed by running the program on each of the 2^n different inputs and remembering the outputs observed on each of the different inputs. This naive algorithm has exponential time and exponential space complexity. The same computation can be actually carried out in polynomial additional space by using nested iteration. The outer iteration ranges over all possible outputs and checks if

there is an input that leads to that particular output by iterating over all possible inputs. Hence, in this alternative algorithm, the program has to be run 2^{2n} times. The latter observation immediately leads to the result that the decision problem of checking whether information leaked in non-recursive boolean programs is equal to (or less than) a given rational number in PSPACE. Indeed, the latter algorithm provides an immediate polynomial-time reduction to the problem of checking whether the program counter reaches a given location in a program, which is also known to be PSPACE-complete.

Now, both the above algorithms for computing information leaked have exponential running time and hence they do not scale very well as n, the number of input bits increase. However, it has been suggested in [1] that one can potentially exploit the reduction to the reachability problem in order to estimate the amount of information leaked by using modelchecking tools as modelchecking tools were originally developed for checking whether a particular state in a program is reachable (on any possible input).

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 The problem and two intuitive solutions

We define a program P as follows:

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* $O \in \{0, 1, 2, ..., 1 \ll bitLength - 1\}$.

And we need to count the number of outputs of P. We come up with two approaches:

1. Put the program in a double loop and count the number of outputs. The outer loop is for possible outputs and the inner loop is for inputs. When the program in the inner loop produces an output which matches the outer loop, counter increases.

2. Let the program iterate through all input values and record the output hit results in a bit array. Counter increases when a bit flips.

The first approach is time-consuming, while the second one is memory-consuming. We will discuss these two approaches in detail in the subsections.

2.3.1 Double loop

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 this output, *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.

In this approach, 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

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
```

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 our 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.

Bit length	Time(s)	Multiplier
14	0.708	
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.

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.

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

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

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

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.

2.3.3 Comparison and our choice

Table 2.2 and 2.3 show a comparison on execution time and memory usage respectively for these two approaches. The single loop and array approach has exponential growth for both time and memory, while the double loop approach has exponential growth for time but linear growth for memory, so we choose the double loop approach.

	Execution time	Growth
Double loop	$2^{2\times bitLength} \times t(P)$	$\times 4$
Single loop & array	$2^{bitLength} \times t(P)$	$\times 2$

Table 2.2: Comparison of the two approaches on execution time and their growth. $t(P)$ is the execution time of the program within the loop.

	Memory requirement(bits)	Growth
Double loop	$7 \times bitLength + 1$	$+7$
Single loop & array	$2^{bitLength} + 7 \times bitLength + 1$	$+2^{bitLength} + 7$

Table 2.3: Comparison of the two approaches on memory requirement and their growth.

Chapter 3

Semi-monotonic programs

In this chapter we describe our discovery that if the test program P meets a certain property, then we can use a different wrapping loop and greatly reduce the time Getafix and jMoped needs to check for reachability.

3.1 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 3.1 shows the C code we used.

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

```
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;
}       }       }
```

Listing 3.1: Initial implementation of the double loop with sanity check in C.

the upper bound is not tested. Second, in the inner loop, we did not initialize S 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 $[0, n]$.

3.2 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 3.2, we started with monotonicity and rearranged the architecture of the outer pro-

gram. The problem is to count the number of outputs of a monotonically increasing function, and our solution is to iterate through the entire range of S and keep track of the highest 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. Set the range of n to $[0, 2^{bitLength} - 2]$.
2. Calculate $P(S_n)$ to be O_n and store it in set $O[]$.
3. Calculate $P(S_{n+1})$ to be O_{n+1} has to be either:
   (a) Greater than O_n, in which case we store O_{n+1} in $O[]$.
   (b) Already in set $O[]$.
4. Continue step 2 on the next n value until n reaches its upper bound.

This is wrong. Correct this.

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.

```
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;
}
```

Listing 3.2: Implementation of the optimization with sanity check in C.

Chapter 4

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 [**?**], Berkeley Lazy Abstraction Software Verification Tool (Blast) from [**?**], Getafix from [**?**] 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
```

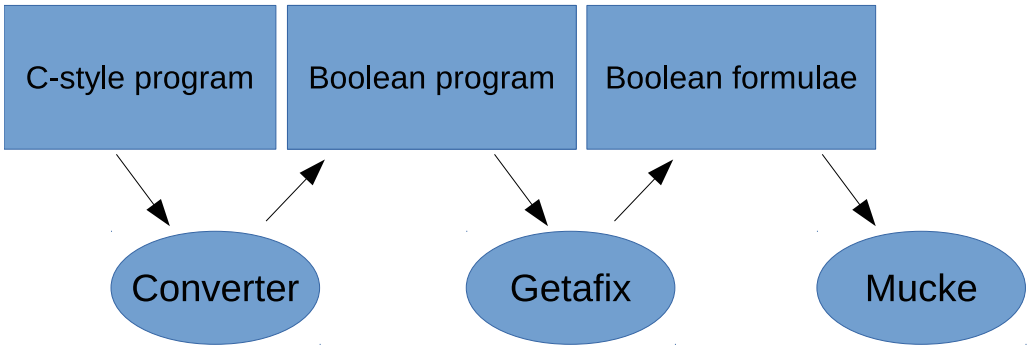

Figure 4.1: Workflow of calculating information leakage with Getafix.

4.1 Getafix

Getafix is a symbolic model checker for Boolean programs implemented in [**?**]. 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 [**?**].

Mostly copied from Getafix website. Need rewrite

4.1.1 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 4.1 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 ^, 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.

Table 4.1: 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);

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:

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 4.1 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 4.1 also contains an example of how the parser deals with a high-level operator.

3. In the boolean code syntax which Getafix defines, function call plus the semicolon 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 4.1 shows a sample function by the generator.

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.

```
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
```

Listing 4.1: Greater than operator as a function in boolean program with bit length of 2.

4.2 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 rewrite our tests in Java so that we can use jMoped on them.

need more explanation

4.3 Tests and results

Before we have the converter, we coded a few test cases manually and Getafix gave us the correct answers. 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. As with jMoped, we rewrite the tests in Java. We decide to run the eight test cases from paper [**?**]. In each test, O represents the output of the test program, and S is the input. Also in the tables, optimization refers to the final optimization in the

previous chapter, and an empty cell means that we did not do this experiment.

We did our tests on a laptop computer, and key hardware specifications are:

Model Lenovo ideapad Y580-IFI

CPU Intel Core i5-3210M @ 2.5GHz

Memory 4GBytes DDR3-1600 $\times$ 2

And the software specifications are:

OS Ubuntu 14.04 LTS 32-bit

Getafix Version information not available. Source code retrieved on 2014/4/10

Mucke Version 0.4.4

Eclipse Eclipse juno sr2

jMoped Version 2.0.2

For Getafix, we time the entire process from converter to Mucke using the time command in bash and record the elapsed wall time. For jMoped, we record the elapsed wall time which jMoped reports.

4.3.1 Sanity check

Listing 4.2 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. We can use the optimization in this test, and the timing results are in Table 4.2.

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

Listing 4.2: Sanity check test program.

bit length(bit)	Getafix(s)	Getafix optimized(s)	jMoped(s)	jMoped optimized(s)
6	41.264		9.322	
7	235.64	3.3775	62.074	0.53
8	>600	6.448	326.122	0.875
9		13.491		1.69
10		27.479		3.045

Table 4.2: Timing results for sanity check.

4.3.2 Implict flow

Listing 4.3 shows the code. This test copies the value of S to O indirectly through the if statement when S is less than 7. For other S values, O is 0. The program has 7 different outputs, ranging from 0 to 6. We can use the optimization in this test, and the timing results are in Table 4.3.

```
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;}
        }       }       }       }       }       }
```

Listing 4.3: Implict flow test program.

bit length(bit)	Getafix(s)	Getafix opt(s)	jMoped(s)	jMoped opt(s)
6	58.834		11.865	
7	289.919	8.2645	63.555	0.575
8	>600	22.6435	325.755	0.965
9		61.464		1.86
10		182.1295		3.57

Table 4.3: Timing results for implicit flow.

4.3.3 Mix and duplicate

Listing 4.4 shows the code. This test first calculates the XOR value of the two halves of S (mix) and second duplicates this XOR value twice in O (duplicate). The output count depends on the bit length, and at 8 bits, it has $2^4 = 16$ different outputs. We can use the optimization in this test, and the timing results are in Table 4.4.

```
O = ((S >> 4) ^ S) & 15;
O = O | O << 4;
```

Listing 4.4: Mix and duplicate test program at 8 bits.

bit length(bit)	Getafix(s)	Getafix opt(s)	jMoped(s)	jMoped opt(s)
4	2.7185	0.9825	1.14	0.375
6	44.728	3.719	32.94	0.63
8	>600	32.931	eclipse terminates 1	1.97
10		Memory allocation failed		9.765

Table 4.4: Timing results for mix and duplicate.

4.3.4 Masked copy

Listing 4.5 shows the code. In this test, O is S with its lower half set to 0, or "masked" out. The output count depends on the bit length, and at 8 bits, it has

$2^4 = 16$ different outputs. We can use the optimization in this test, and the timing results are in Table 4.5.

```
O = S & 240;
```

Listing 4.5: Masked copy test program at 8 bits.

bit length(bit)	Getafix(s)	Getafix opt(s)	jMoped(s)	jMoped opt(s)
4	1.728	0.7515	0.605	0.23
6	38.2435	1.3855	10.29	0.345
8	>600	5.254	320.87	0.81
10		39.1355		2.445

Table 4.5: Timing results for masked copy.

4.3.5 Binary search

Listing 4.6 shows the code. This test leaks the upper half of S to O through binary search. The output count depends on the bit length, and at 8 bits, it has $2^4 = 16$ different outputs. We can use the optimization in this test, and the timing results are in Table 4.6.

```
if(O + 128 <= S){O = O + 128;}
if(O + 64 <= S){O = O + 64;}
if(O + 32 <= S){O = O + 32;}
if(O + 16 <= S){O = O + 16;}
```

Listing 4.6: Binary search test program at 8 bits.

bit length(bit)	Getafix(s)	Getafix opt(s)	jMoped(s)	jMoped opt(s)
4	3.3590	1.094	0.905	0.330
6	136.0740	5.947	28.255	0.635
8	>600	65.235	eclipse error	2.315
10		>600		13.530

Table 4.6: Timing results for binary search.

4.3.6 Electronic purse

Listing 4.7 shows the code. Assume S is the account balance, we set the deduction to 5, and output O represents the number of times one can debit such an amount. In this test we set $SMax$ to 19,and the program has 4 outputs, ranging from 0 to 3. We can use the optimization in this test, and the timing results are in Table 4.7.

```
O = 0;
while(S >= 5){
        S = S - 5;
        O = O + 1;
}
```

Listing 4.7: Electronic purse test program.

bit length(bit)	Getafix(s)	Getafix opt(s)	jMoped(s)	jMoped opt(s)
6	10.411	2.7805	1.7	0.305

Table 4.7: Timing results for electronic purse.

4.3.7 Sum query

Listing 4.8 shows the code. This test leaks the sum of its three inputs to O. We set $S1, S2$ and $S3$ to be less than 10, so the program has 28 outputs ranging from 0 to 27. We can use the optimization in this test, and the timing results are in Table 4.8.

```
O = S1 + S2 + S3;
```

Listing 4.8: Sum query test program.

bit length(bit)	Getafix(s)	Getafix opt(s)	jMoped(s)	jMoped opt(s)
6	225.3995	33.488	21.115	2.595

Table 4.8: Timing results for sum query.

4.4 Results summary

From the seven test programs and timing results, we can conclude the following points:

1. At the same bit length, jMoped is faster than Getafix.

2. The optimization can reduce execution time for jMoped and Getafix greatly.

3. Even with the optimization, using a model checker is still slower than executing the double loop directly.

Chapter 5

Conclusion and future works

In this thesis we showed a novel approach of using model checking tools to compute information leakage. The principal idea is to wrap the program to be tested in a loop which counts the number of outputs it has, and instead of directly executing the whole code, we append an if statement with the counter as its condition and apply a model checking tool to check for reachability of the statement within if. We think that this approach may be faster than direct execution.

Our main work divides into three parts:

1. We wrote a converter that converts C-style code into boolean program, which is the input Getafix requires. Later we used the converter on all the seven test cases and it saved us a lot of time on coding the tests.

2. We came up with an optimization method which can greatly reduce the time needed to calculate information leakage, either through model checking tools or through direct execution.

3. We tested the seven programs on both Getafix and jMoped with different bit lengths.

We found that our approach could not run faster than direct execution.

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] R. Chadha, D. Kini, and M. Viswanathan. Quantitative information flow in boolean programs. In *Principles of Security and Trust*, pages 103–119, 2014.

[2] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[3] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[4] J. W. Gray III. Toward a mathematical foundation for information flow security. In *IEEE Symposium on Security and Privacy*, pages 21–35, 1991.

[5] J. K. Millen. Covert channel capacity. In *IEEE Symposium on Security and Privacy*, pages 60–66, 1987.

[6] J. C. Reynolds. Syntactic control of interference. In *POPL '78*, pages 39–46, 1978.

[7] G. Smith. On the foundations of quantitative information flow. In *FOSSACS '09*, pages 288–302, 2009.

[8] P. Černý, K. Chatterjee, and T. A. Henzinger. The complexity of quantitative information flow problems. In *CSF '11*, pages 205–217, 2011.

[9] H. Yasuoka and T. Terauchi. On bounding problems of quantitative information flow. In *ESORICS '10*, pages 357–372, 2010.

[10] H. Yasuoka and T. Terauchi. Quantitative information flow as safety and liveness hyperproperties. In *QAPL 2012*, pages 77–91, 2012.