

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 Dr. Gennaro Parlato for providing instructions on Getafix, Truc Nguyen Lam for providing an executable of Interproc, and Umang Mathur for providing MOPED source code and building instructions. Their help 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

ACKNOWLEDGMENTS	ii
LIST OF TABLES	v
LIST OF FIGURES	vi
ABSTRACT	vii
CHAPTER	
1 Introduction	1
2 Theoretical background	7
2.1 Min-entropy and information leakage in programs	7
2.2 The problem and two intuitive solutions	9
2.2.1 Double loop	10
2.2.2 Single loop and array	11
2.2.3 Comparison	13
3 Semi-monotonic programs	14
3.1 Discovery by mistake	14
3.2 Improvements	17
4 Experiment with Getafix and jMoped	20
4.1 Getafix	21
4.1.1 The converter	22
4.2 jMoped	25

4.3	Tests and results	26
4.3.1	Sanity check	27
4.3.2	Implicit flow	27
4.3.3	Mix and duplicate	28
4.3.4	Masked copy	29
4.3.5	Binary search	30
4.3.6	Electronic purse	31
4.3.7	Sum query	31
4.4	Results summary	32
5	Conclusion and future works	33
APPENDIX		
A	Context-free grammar for the converter	35
BIBLIOGRAPHY		38

LIST OF TABLES

Table		Page
2.1	Execution time of an empty double loop with different bit length. . .	12
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.	13
2.3	Comparison of the two approaches on memory requirement and their growth.	13
4.1	Examples of input and output of the parser, with bit length of 4 . . .	23
4.2	Timing results for sanity check.	27
4.3	Timing results for implicit flow.	28
4.4	Timing results for mix and duplicate.	29
4.5	Timing results for masked copy.	30
4.6	Timing results for binary search.	30
4.7	Timing results for electronic purse.	31
4.8	Timing results for sum query.	32

LIST OF FIGURES

Figure		Page
3.1	A function that follows the properties from the first optimization. . .	16
3.2	A function that follows the properties from the second optimization. .	18
4.1	Workflow of calculating information leakage with Getafix.	21

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 reachability in programs. Thus we propose a hypothesis that we can estimate leakage using model checking tools which are originally developed for checking reachability.

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 semi-monotonicity then we can use a different reduction to the problem of checking reachability. We observe a dramatic improvement in performance with this new reduction.

Chapter 1

Introduction

A desirable property for a program is *non-interference* [1, 2] which informally says that a program should never leak any information about its secret inputs. Formally, *non-interference* [1, 2] says 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 [3, 4, 5, 6] 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 [6]. 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 [7, 8, 9, 10]. 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. Since different inputs can lead to different outputs, this naive algorithm can take 2^n -additional space. This naive algorithm (which we shall call algorithm A for the rest of the section) 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 (which we shall call algorithm B for the rest of the section) 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, algorithm A for computing min-entropy takes above takes at least exponential time and exponential space, while algorithm B takes at least exponential time. Hence, they do not scale very well as n , the number of input bits increase. However, it has been suggested in [9] that one can potentially exploit the polynomial-time 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). These modelchecking tools do not explicitly run the program on all inputs. Instead use a variety of heuristics to solve the reachability problem.

We tested the above hypothesis using two popular model-checking tools, jMoped [11] and Getafix [12]. Jmoped is a tool developed for checking reachability in Java programs. The key technology used in JMoped is the use of Binary Decision Diagrams (BDDs) [13, 14]. BDDs are data structures designed to efficiently store Boolean functions. In jMoped, the input program is translated as set of *transitions* on the states

of the program (a state of a non-recursive program is the current line number and the values of the variables of the program). Each transition then models how program execution changes the state of the program. However, instead of writing transitions explicitly, BDDs are used to store the set of transitions. Reachability can then be encoded as the least fix-point solution of a set of Boolean equations which can be solved with efficient BDD operations. Getafix [12] is based on similar ideas except that it accepts only Boolean programs and reachability is encoded as a winning condition on a 2-player game on a transition system. Although, these tools should work in theory, our tests for estimating min-entropy indicate that these do not scale very well as the number of bits in the input increase.

However, we identified a condition under which there is a dramatic improvement in the performance of these tools. In particular, we show that if the program P whose information leakage we are estimating satisfies the additional property of *semi-monotonicity* then the computation of min-entropy becomes more feasible. 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. Recall that the function P_{func} is monotonically increasing if for each pair of n -bit binary numbers s_1, s_2 such that $s_1 \leq s_2$, we have that $P_{func}(s_1) \leq P_{func}(s_2)$. Note that a monotonically increasing function we have that for each n -bit binary number s , $P_{func}(s) \geq \max_{s' \leq s} P_{func}(s')$. We say the the program P is semi-monotonically increasing if for each n -bit binary number s either $P_{func}(s) \geq \max_{s' \leq s} P_{func}(s')$ or $P_{func}(s) \in \{P_{func}(s') \mid s' \leq s\}$. We can similarly define semi-monotonically decreasing programs. The program P is said to be semi-monotonic if P is either semi-monotonically increasing or decreasing.

The key observation that we exploit is that for semi-monotonicity we can essentially use algorithm A for computing information leakage except that we do not need to use exponential additional space. Instead, if the program P is semi-monotonically increasing (increasing respectively) then while iterating over the inputs, we just need to remember in each iterative step the total number of the distinct outputs seen thus far as well as the highest (lowest respectively) output seen thus far. Thus, for semi-monotonic programs we have a new polynomial-time reduction of the decision problem of checking whether information leaked in non-recursive boolean programs is equal to (or less than) a given rational number to the problem of checking whether a line number in a program is reachable or not. We now use this new reduction to estimate the information leaked in programs using jMoped and Getafix and observe a dramatic improvement in their performance with this new reduction.

Related work. The *complexity* of computing the amount of leakage in Boolean programs has been considered recently in [15, 7, 8, 10, 16, 10, 9]. In recent years, several automated approaches from model checking [17, 18, 19, 20], static analysis [21, 22, 23, 17], and statistical analysis [18, 24] have been employed to compute information leakage. We mention the most closely related work.

[25, 26] estimates min-entropy leakage using SMT solvers. SMT solvers are tools developed to check whether a given first-order formula over a decidable first-order theory has a solution or not. In [25, 26], the authors estimate an upper bound of number of feasible outputs of a program by iterating over each pair of output bits and computing how many different values of these output bits can be achieved (i.e., how many of the values $\{00, 01, 11, 10\}$ can be actually observed). This technique only

yields an upper bound, and it is easy to construct examples where the upper bound is a very poor estimate of the actual value.

Another line of closely related work is the work on computing information leakage measure using Shannon entropy [17, 27]. In this line of work, Shannon entropy (and not min-entropy) is used to measure uncertainty of the adversary. Usually an assumption of uniformly distributed inputs is made. When Shannon entropy is used to measure information leakage, one has to compute not only the number of feasible outputs but one also has to compute, for each feasible output, the number of inputs that lead to that particular output. In order to compute these two things, usually an equivalence relation on inputs is defined as follows: two inputs are equivalent if they lead to the same output. Then one needs to compute these equivalence classes. In [17], these equivalence classes are constructed iteratively by first starting with a single equivalence class and progressively refining them until they can be refined no further. At each iteration, the equivalence relation is characterized using logical formulas and the refinement step uses experimental runs. In [18], statistical analysis is used for this construction. In [27], bounded model-checking is used to compute the equivalence. In bounded model-checking, reachability is checked assuming that the program executes at most a bounded number of steps. Hence, this method gives an approximate value of information leakage.

Chapter 2

Theoretical background

2.1 Min-entropy and information leakage in programs

We briefly recall the mathematical theory behind computation of information leakage in programs. For computing information leakage, a program is usually considered as an information-theoretic channel between its inputs and outputs. Formally,

Definition 1. *An information-theoretic channel \mathcal{C} is a triple (S, O, C_{SO}) such that*

- *S is a finite set of secret input values,*
- *O is a finite set of secret output values, and*
- *C_{SO} is a $|S| \times |O|$ non-negative matrix such that $\sum_{o \in O} C_{SO}(s, o) = 1$.*

\mathcal{C} is said to be deterministic if for each $s \in S$ there is a unique $o \in O$ such that $C_{SO}(s, o) = 1$.

For $s \in S$ and $o \in O$, the quantity $C_{SO}(s, o)$ is the conditional probability of obtaining output o given that the input to the channel is s . Given any information-theoretic channel $\mathcal{C} = (S, O, C_{SO})$ and *a priori* distribution $Prob_S$ on S , we get a joint probability distribution $Prob_{S,O}$ on $S \times O$ given as $Prob_{S,O}((s, o)) = Prob_S(s)C_{SO}(s, o)$. For the purpose of this thesis, we shall associate with each deterministic program P , a deterministic channel $P_{\mathcal{C}} = (S, O, C_{SO})$ where S is the set of all possible secret inputs of P , O is the set of all possible outputs of P and $C_{SO}(s, o) = 1$ iff the program P outputs o on input s .

When measuring information leakage in programs using min-entropy [6], an adversary is considered which tries to guess the input to the program before and after the program is executed. The difference in *uncertainty* about S before and after the program execution is taken to be the amount of information leaked. It was proposed in [6] that min-entropy be used as the measure of uncertainty. We refer the reader to [6] for details regarding theoretical foundations behind min-entropy. We just point out here the relevant formulas.

Definition 2. *Let $\mathcal{C} = (S, O, C_{SO})$ be an information-theoretic channel and let $Prob_S$ be an a priori distribution on S . The initial uncertainty of the adversary, written $H_{\infty}(S)$, is taken to be $-\log_2 \max_{s \in S} Prob_S(s)$ and final uncertainty, written $H_{\infty}(S|O)$, is taken to be $-\log_2 \sum_{o \in O} \max_{s \in S} Prob_{S,O}((s, o))$. The amount of information leaked by \mathcal{C} , written $L_{S,O}$ is*

$$L_{S,O} = H_{\infty}(S) - H_{\infty}(S|O).$$

As expected, the amount of information leaked by a program is said to be the amount of information leaked by the channel $P_{\mathcal{C}}$ associated to it. We are interested in computing the maximum possible amount of leakage under all possible a priori

distributions (this quantity is also known as *min-capacity*). It turns out that for deterministic programs, this just amounts to computing the number of outputs that are actually feasible, i.e., are actually realized by some input.

Theorem 1. ([6]) *Let $\mathcal{C} = (S, O, C_{SO})$ be a deterministic channel. Let $feasible = \{o \in O \mid \exists s \in S. C_{SO}(s, o) = 1\}$. Then for any apriori distribution $Prob_S$ on S , we have that the information leaked $L_{S,O} \leq \log_2 |feasible|$. Furthermore $L_{S,O} = \log_2 |feasible|$ if $Prob_S$ is the uniform distribution on S .*

Remark. *Sometimes, Shannon entropy is used to measure uncertainty instead of min-entropy. However, it was shown in [6], that if we measure information leaked using Shannon entropy then Shannon capacity, the maximum amount of information leaked, of deterministic programs under all possible apriori distributions matches exactly the min-capacity.*

2.2 The problem and two intuitive solutions

We define a program P as follows:

Definition 3. *Let $bitLength \in \{0, 1, 2, \dots, 32\}$ and let P be a function with one input and one output. Also input $S \in \{0, 1, 2, \dots, 2^{bitLength} - 1\}$ and output $O \in \{0, 1, 2, \dots, 2^{bitLength} - 1\}$.*

We need to count the number of feasible outputs of P in order to compute the information leakage. We have two approaches to this problem:

1. Put the program in a double loop and count the number of outputs. The outer loop iterates over possible outputs and the inner loop iterates over possible

inputs. When the program in the inner loop produces an output which matches the outer loop, the counter counting the number of feasible outputs increases.

2. Let the program iterate through all input values and record the output hit results in a bit array. The 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.

2.2.1 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 $7 \times \textit{bitLength} + 1$ at $O(\textit{bitLength})$. If we assume program *P* has time complexity of $O(t(P))$ then the total execution time for the double loop when break is never reached is $2^{\textit{bitLength}} \times 2^{\textit{bitLength}} \times O(t(P))$. Thus the time complexity is $2^{O(\textit{bitLength})} \times O(t(P))$.

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 $2^{32} \times 10.30$

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
```

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 $2^{2 \times n} \times O(t(P))$. 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.

2.2.2 Single loop and array

In Algorithm 2, we create a bit array with size equal to the maximum number of possible outputs ($1 \ll \text{bitLength}$, or $2^{\text{bitLength}}$) and initialize it with zeros. While we

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.

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.

```

 $S \leftarrow 0$ 
 $O \leftarrow 0$ 
 $SIn \leftarrow 0$ 
 $OOut \leftarrow 0$ 
 $OCounter \leftarrow 0$ 
 $SMax \leftarrow 1 \ll bitLength - 1$ 
 $OMax \leftarrow 1 \ll bitLength - 1$ 
 $OHit[OMax + 1] \leftarrow [0]$ 
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 $7 \times bitLength + 1$ 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 $2^{O(bitLength)}$. The time complexity for this algorithm is $(2^{O(bitLength)}) \times O(t(P))$.

	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.

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 about 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.

2.2.3 Comparison

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.

Chapter 3

Semi-monotonic programs

In this chapter we describe our discovery that if the test program P meets a certain property that we call semi-monotonicity, then we can use a different wrapping loop.

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. We set the base to 0.

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 $2^{2 \times 32}$ times.

We found that removing the initialization of S to 0 can be a way to speed up some programs, but not all. Specifically, program P has to obey certain properties in order for this initialization to give an accurate answer:

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

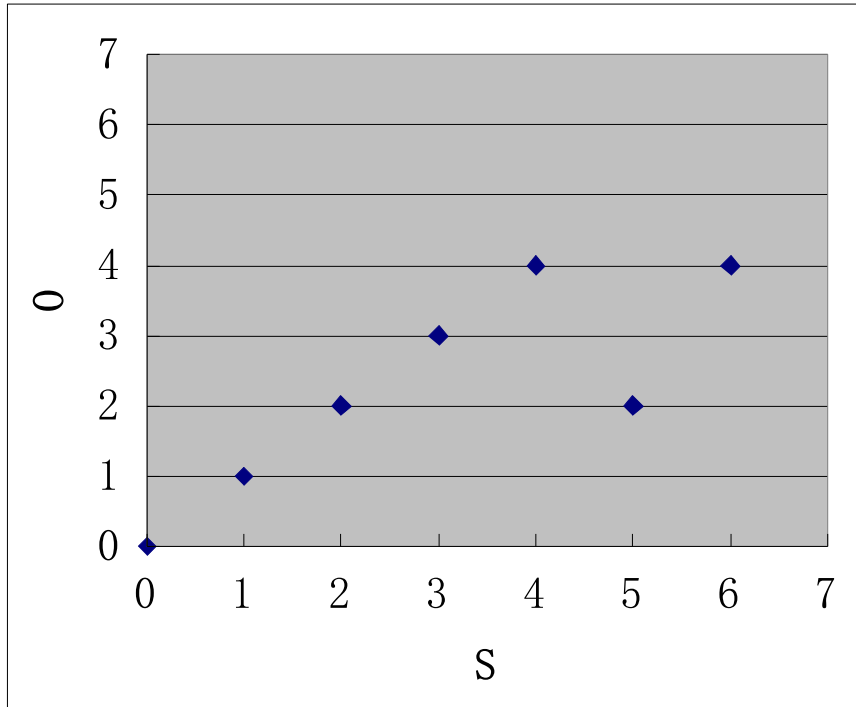


Figure 3.1: A function that follows the properties from the first optimization.

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 $[n + 1, 1 \ll \text{bitLength} - 1]$ can be produced by at least one S in range $[0, n]$.

Figure 3.1 shows an example function that follows these properties. In the first part $[0, 4]$, function output has to start from 0 and be continuously increasing. As to the second part $[5, 6]$, each output has to be equal to an output in $[0, 4]$.

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 as shown in Listing 3.2 to improve this optimization method, 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 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 different from the first one.

For program P , it has to follow either of the following conditions:

1. $P_{func}(s) \geq \max_{s' \leq s} P_{func}(s')$
2. $P_{func}(s) \in \{P_{func}(s') \mid s' \leq s\}$

To enable the optimization.

Also Figure 3.2 shows an example function that follows these conditions. A new output value has to be either larger than all previous values like when S is 3, or is within the set of all previous outputs like when S is 6.

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

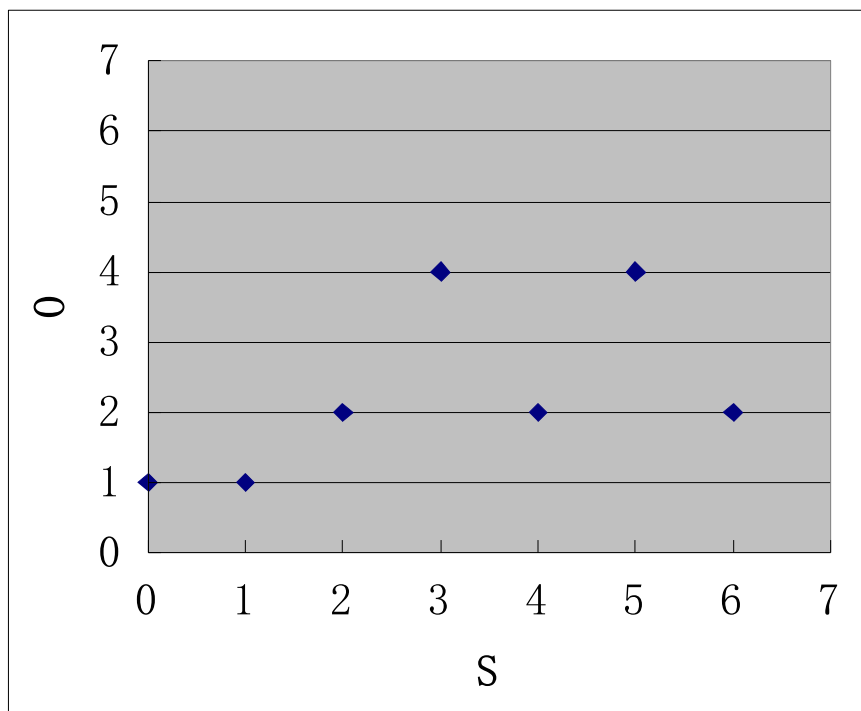


Figure 3.2: A function that follows the properties from the second optimization.

$2^{\text{bitLength}} \times t(P)$, which is $1/2^{\text{bitLength}}$ of the execution time of the original double loop.

```
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. Model-checking tools can help us to decide if a particular line in a program can be reached. Our approach entails appending 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 constraints in the condition block. In our tests, we used only an equality test. Our purpose is to see our optimization performs with respect to other optimizations and how the tools scale with the number of bits. However, note we can easily convert reachability tests into an exact computation by using a binary search to successively refine the number of the possible values of the counter which counts the number of outputs.

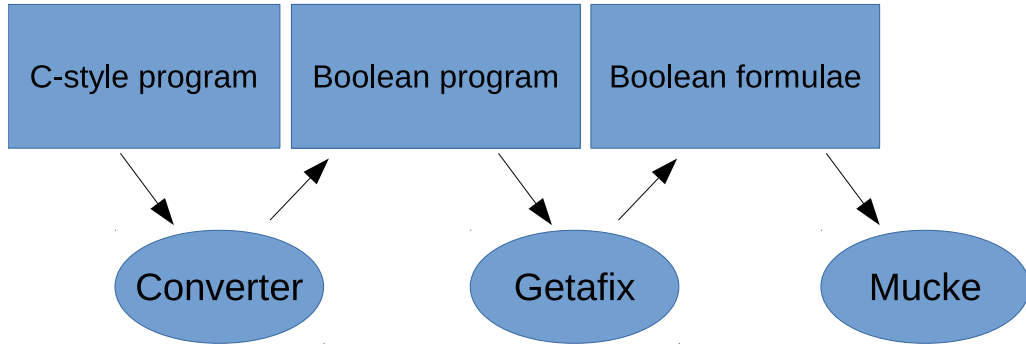


Figure 4.1: Workflow of calculating information leakage with Getafix.

We experiment on several model-checking tools, including Interproc from [28], Berkeley Lazy Abstraction Software Verification Tool (Blast) from [29], Getafix from [12] and jMoped from [30]. We can not get correct reachability results from Interproc and Blast, so we shift our focus onto Getafix and jMoped.

<p>Algorithm 3: Determine if <i>OCounter</i> meets certain constrains.</p>

<pre> if value of <i>OCounter</i> meets certain constrains then reach: <i>OCounter</i> // a label followed by a statement end if </pre>
--

4.1 Getafix

Getafix is a symbolic model checker for Boolean programs that supports reachability checking. It translates sequential Boolean programs into Boolean modal mu-calculus formulae and uses the model-checker Mucke [31] to solve the reachability problem. Mucke itself model-checks symbolically using Boolean Decision Diagrams.

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 $2^{bitLength} - 1$.
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.

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);

Table 4.1: Examples of input and output of the parser, with bit length of 4

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.

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 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 4.1 shows a sample function by the generator.

```
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.

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.

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.

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 [25]. In each test, O represents the output of the test program, and S is the input. Also in the tables, opt refers to the final optimization in the previous chapter, and we set the maximum execution time to 600 seconds.

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 junos 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 opt(s)	jMoped(s)	jMoped opt(s)
6	41.264	2.093	9.32	0.42
7	235.640	3.378	62.07	0.53
8	>600	6.448	326.12	0.88
9		13.491	JVM terminates	1.69
10		27.479		3.05
11		61.871		6.23
12		140.115		14.80
13		302.96		35.17
14		580.982		78.29
15		>600		181.82
16				444.17
17				JRE Error

Table 4.2: Timing results for sanity check.

4.3.2 Implicit 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: Implicit flow test program.

bit length(bit)	Getafix(s)	Getafix opt(s)	jMoped(s)	jMoped opt(s)
6	58.834	2.937	11.87	0.47
7	289.919	8.265	63.56	0.58
8	>600	22.644	325.76	0.97
9		61.464	>600	1.86
10		182.130		3.57
11		429.298		7.21
12		>600		15.75
13				36.31
14				81.53
15				182.01
16				415.69
17				>600

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	Getafix(s)	Getafix opt(s)	jMoped(s)	jMoped opt(s)
4	2.719	0.983	1.14	0.38
6	44.728	3.719	32.94	0.63
8	>600	32.931	JVM terminates	1.97
10		Memory alloc error		9.77
12				69.01
14				JVM terminates

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	Getafix(s)	Getafix opt(s)	jMoped(s)	jMoped opt(s)
4	1.728	0.752	0.61	0.23
6	38.244	1.386	10.29	0.35
8	>600	5.254	320.87	0.81
10		39.136	JVM terminates	2.45
12		247.135		11.78
14		>600		57.08
16				327.95
18				JRE fatal error

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	Getafix(s)	Getafix opt(s)	jMoped(s)	jMoped opt(s)
4	3.359	1.094	0.91	0.33
6	136.074	5.947	28.26	0.64
8	>600	65.235	JVM terminates	2.32
10		>600		13.53
12				99.49
14				JVM terminates

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)
5	10.411	2.781	1.70	0.31

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)
5	225.400	33.488	21.12	2.60

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. The best result we get is using jMoped with the optimization, but still the time growth appears to be exponential.

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. Given the success of model-checking to program analysis, 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 in 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. The optimization works on certain programs that satisfy special properties which we call semi-monotonicity.

3. We tested seven benchmarks from [6] on both Getafix and jMoped with varying bit lengths.

We found that a direct approach in combination with the model checkers does not scale well. We managed to reach 16 bits with jMoped and the optimization in most of the tests, but a real-world program would typically use 64 bits or more.

Potential future work includes trying more sophisticated model checkers. Another approach would be to see if the model-checking algorithms used in jMoped and Getafix can be modified to count the number of outputs. Another possible direction is to design heuristics to check whether programs are semi-monotonic or not in order to see if our optimization can be applied. Currently, we check for semi-monotonicity by hand.

Appendix A

Context-free grammar for the converter

This is the converter input grammar. We take it out of the parser and remove the actions to make it serve as a reference.

```
%start program
%%
program:
    var_decls stmts
var_decls:
    %empty
    | var_decls var_decl
var_decl:
    T_VAR T_ID ';'
    | T_VAR T_ID '=' exp ';'

```

```

stmts:
    %empty
    | stmts stmt

stmt:
    T_ID ':' stmt
    | exp ';'
    | T_ID '=' exp ';'
    | T_IF '(' exp ')' block T_ELSE block
    | T_IF '(' exp ')' block
    | T_WHILE '(' exp ')' block
    | T_GOTO T_ID ';'

block:
    '{' stmts '}'

exp:
    T_NUM
    | '(' exp ')'
    | T_ID
    | exp '+' exp
    | exp '^' exp
    | exp '-' exp
    | exp '|' exp
    | exp '&' exp
    | exp "<<" exp
    | exp ">>" exp

```

| exp '<' exp
| exp "<=" exp
| exp '>' exp
| exp ">=" exp
| exp "==" exp
| exp "!=" exp

Bibliography

- [1] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
- [2] J. C. Reynolds. Syntactic control of interference. In *POPL '78*, pages 39–46, 1978.
- [3] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, 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] G. Smith. On the foundations of quantitative information flow. In *FOSSACS '09*, pages 288–302, 2009.
- [7] H. Yasuoka and T. Terauchi. On bounding problems of quantitative information flow. In *ESORICS '10*, pages 357–372, 2010.
- [8] H. Yasuoka and T. Terauchi. Quantitative information flow as safety and liveness hyperproperties. In *QAPL 2012*, pages 77–91, 2012.

- [9] R. Chadha, D. Kini, and M. Viswanathan. Quantitative information flow in boolean programs. In *Principles of Security and Trust*, pages 103–119, 2014.
- [10] P. Černý, K. Chatterjee, and T. A. Henzinger. The complexity of quantitative information flow problems. In *CSF '11*, pages 205–217, 2011.
- [11] D. Suwimonteerabuth, F. Berger, S. Schwoon, and J. Esparza. jMoped: A test environment for Java programs. In Werner Damm and Holger Hermanns, editors, *Proceedings of CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 164–167. Springer, July 2007. Tool paper.
- [12] S. La Torre, P. Madhusudan, and G. Parlato. Analyzing recursive programs using a fixed-point calculus. In *PLDI*, pages 211–222, 2009.
- [13] C. Y Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.
- [14] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [15] H. Yasuoka and T. Terauchi. Quantitative information flow - verification hardness and possibilities. In *CSF '10*, pages 15–27, 2010.
- [16] R. Chadha and M. Ummels. The complexity of quantitative information flow in recursive programs. In *FSTTCS*, pages 534–545, 2012.
- [17] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *IEEE Symposium on Security and Privacy*, pages 141–153, 2009.

- [18] B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *CSF '10*, pages 3–14, 2010.
- [19] K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Probability of error in information-hiding protocols. In *CSF '07*, pages 341–354, 2007.
- [20] K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Anonymity protocols as noisy channels. *Information and Computation*, 206(2-4), 2008.
- [21] Da. Clark, S. Hunt, and P. Malacaria. Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science (Proc. QAPL '04)*, 112:49–166, 1984.
- [22] D. Clark, S. Hunt, and P. Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic Computation*, 15(2):181–199, 2005.
- [23] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.
- [24] K. Chatzikokolakis, T. Chothia, and A. Guha. Statistical measurement of information leakage. In *TACAS '10*, pages 390–404, 2010.
- [25] Z. Meng and G. Smith. Calculating bounds on information leakage using two-bit patterns. In *PLAS '11*, 2011.
- [26] Z. Meng and G. Smith. Faster two-bit pattern analysis of leakage. Proc. QASA 2013: 2nd International Workshop on Quantitative Aspects of Security Assurance, Royal Holloway, University of London, 2013.

- [27] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *ACSAC*, pages 261–269, 2010.
- [28] G. Lalire, M. Argoud, and B. Jeannet. Interproc analyzer. <http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html>.
- [29] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.
- [30] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *TACAS*, LNCS 3920, pages 489–503, 2006.
- [31] Armin Biere. μ cke - efficient μ -calculus model checking. In *Proceedings of CAV, 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 468–471. Springer, 1997.