CONfidentiality CERTifier
A Modeling and Verification Framework for Program Confidentiality

Saeid Tizpaz Niari

University of Colorado-Boulder
saeid.tizpazniari@colorado.edu

Abstract

By July 2013 51% of US adults, or 61% of Internet users, were banking online and out of that 31% of US adults, or 35% of cell phone owners, were banking using their mobile phones[1]. In doing that they put great trust on systems/programs providing banking services to keep their sensitive data private (confidentiality) and in consistent state and to be who they say they are (integrity), and finally to be accessible when required (availability). This *confidentiality*, *integrity*, and *availability* triad is at the heart of program security, where confidentiality of sensitive information is arguably one of the most fundamental security issues. In this work we focus on confidentiality aspect of programs and propose modeling and verification framework for confidentiality certification. We propose the use of finite-state transducers as abstract models of programs and we sketch our tool ConCert based on Z3 theorem prover to certify confidentiality of programs modeled as transducers.

1. Introduction

A program/software is said to be confidential if it does not (directly or indirectly) disclose any secret information to unauthorized individuals, entities, or process. Static analysis of the source-code of the program is an important tool to certify confidentiality, however their direct application is limited due to the presence of so-called side-channels where indirect effects of running the program (such as time, memory, power consumption, electro-magnetic radiations) may leak secret information to third-parties. This poses an extra challenge to the traditional static analysis based approaches to confidentiality verification, since such side-channels are not always explicitly present in the syntax of the program. In our work, we propose using dynamic analysis to infer ghost variables (artificial observable variables) and their updates as function of the state of the program, and then use formal analysis to falsify/verify confidentiality property.

A common approach to enforce confidentiality is via notion of *noninterference* which requires that an adversary should deduce nothing about the secret inputs from observing public inputs. However, one can argue that achieving noninterference is often hard, because oftentimes programs need to reveal information that depends on the secret inputs. For instance, in an election protocol the individual votes should be secret, but of course we want to reveal the tally of votes publicly. A more nuanced approach to confidentiality is to quantify "how much" information is being leaked, and perhaps allowing us to tolerate "small" leaks [Geo09].

In order to study formally and study various definitions of confidentiality properties, we reduce the problem of confidentiality of programs to some interesting properties in finite-state transducers. A finite state transducer (FST) is a 7-tuple, $T = (Q, \Sigma, \Gamma, \delta, \omega, q_0, F)$ where Q is a finite set of states, Σ is the input alphabet, Γ is the output alphabet, $q_0 \in Q$ is the set of initial states, $F \subseteq Q$ is the set of terminal states, $\delta : Q \times \Sigma \to Q$ is the transition function, and $\omega : Q \times \Sigma \to \Gamma$ is the output function. We say that a transducer is functional if there is at most one output for any given input. Another interesting property for transducers is the *k-ambiguity*. We say that a transducer T is k-ambiguous if for every word $w \in \Sigma^*$, T has at most k different runs (sequence of states) in its successful computations. The notion of *K-valuedness* generalizes the idea of functionality and ambiguity – a transducer is called k-valued if it produces at most k distinct outputs for each acceptable input. More formally, T is k-valued if for each input word $w \in \Sigma^*$, the maximum number of distinct set of output words $w' \in \Gamma^*$ is bounded from above by k. Observe that 1-valuedness is simply the functionality. One of the key observation of this work is that the confidentiality property of a program

[1] `http://www.pewinternet.org/2013/08/07/51-of-u-s-adults-bank-online/`

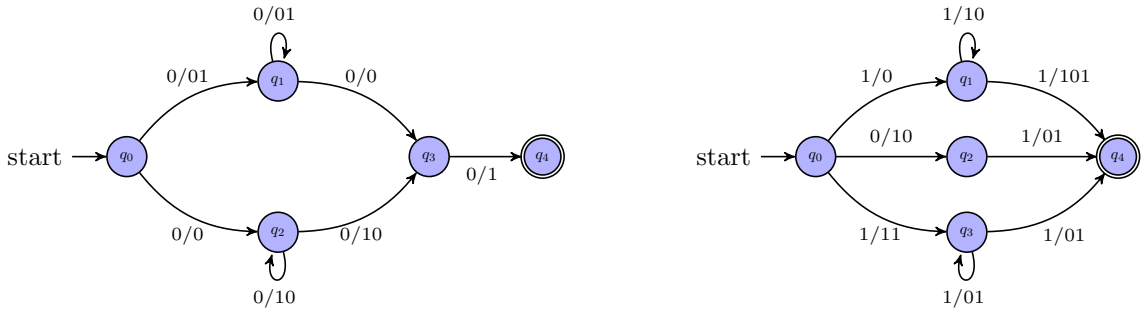

Figure 1: Example of transducers and their properties: (a) a functional transducer (b) a 2-valued transducer

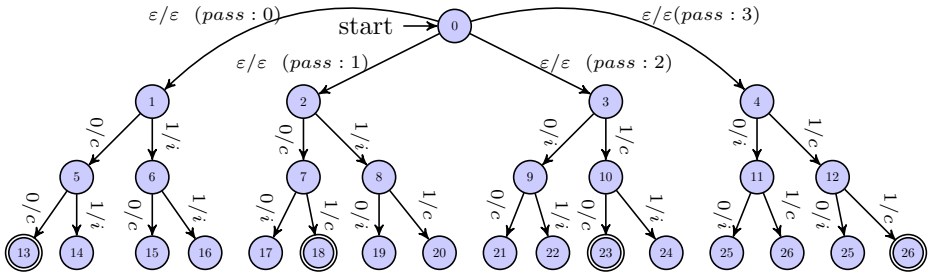

Figure 2: Transducer model of password matcher program: An insecure implementation which is 4-valued

can be reduced to checking k-valuedness for an appropriate transducer model. Under such reduction, the functional transducer corresponds to non-interference.

Example 1. *See Figure 1.(a) that indicates a 2-ambiguous transducer which is functional (1-valued). Given any acceptable input, we always observe the same output. Figure 1.(b) shows 2-valued transducer. There are inputs like 1111 in which we can observe two different outputs like 01010101 and 11010101 in two successful computations.*

In this work, we present an incremental verification framework to decide whether a given transducer is functional (1-valued), and if it is not functional then we quantify how much information is being leaked through finding the biggest k for which the transducer satisfies the k-valued property. To the best of our knowledge, this is the first time to relate transducer properties to confidentiality. In addition, verification of transducer properties using Z3 theorem solver is another contribution.

2. Motivation Example

In this section, we want to discuss motivation examples. We will provide three examples in different domains to illustrate our modeling and verification methods.

Example 2. *Consider a simple 2-bit password checker program. In one implementation (PA), suppose the program has bit-wise comparison of inputs, and it has different behaviors in the case of correct and incorrect bit. For instance, if the input bit matches, it has 2 ms sleep time; otherwise it has 5 ms sleep time. Let's show the behavior for each correct bit with 'c' and for each incorrect bit with 'i' as observable output. We can model this program using transducer which is shown in Figure 2. Now, we can verify k-valued property in this transducer. Given any input like (00), we will observe four different outputs: {cc,ic,ci,ii}. This means that the program is 4-valued. Now, consider another implementation (PB) of the same password checker. In this case, program does not have any different behavior for correct and incorrect bit. Instead, it checks whole input and gives the result whether the input is matched (c), or not (i). This implementation has two different outputs namely {c,i} for any given input, and hence is 2-valued.*

In the modeling of this example, we over-approximate all possible value of secret in the program and show all possible values as different branches (which can be taken non-deterministically) in outgoing transitions

```
int z = random.nextInt(4);
LeakWatchAPI.secret("z", z);
int i = 0, j = 0;

if(z >= 0 && z <=1)
{
    if(B1)
    {
        while(i<n) {i++;}
        LeakWatchAPI.observe("01");
    }
    else
    {
        i = 0;
        while(i<n) {
            while (j < Math.log(n)){
            i++;
            j++;}
        }
        }
        LeakWatchAPI.observe("10");
    }
}
else
{
    if(B1)
    {
        while (i<n) {i++;}
        LeakWatchAPI.observe("01");
    }
    else
    {
        while (i < n){
            while (j < n){
            i++;
            j++;}
        }
        LeakWatchAPI.observe("11");
    }
}
```

Figure 3: A simple java program annotated with observable labels (complexity information) and a 2-bit secret variable (z)

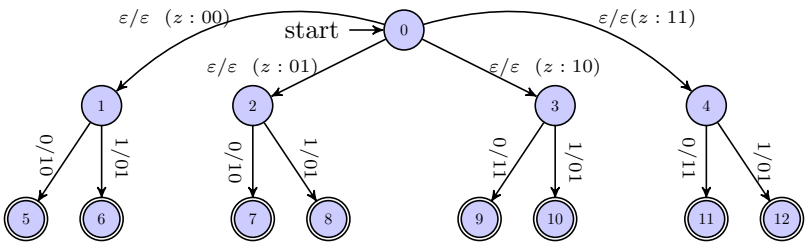

Figure 4: Transducer model of java program in Figure 3. Given B1 is 0 (false), we observer two different output {10,11}

of the initial state (you can see each outgoing transition of initial state is corresponding to a possible value for password). In case of PA, we have 4-valued transducer which is a total leakage for our secret value. The first guess will reduce the uncertainty of observers to zero, and he/she will know the secret password. In case of PB, we have 2-valued transducer which is a normal implementation for password checker. The first guess will reduce the uncertainty of observer by one; however, he/she will still be uncertain about the value of secret after one incorrect guess. We should note that achieving 1-valuedness (non-interference) for password checkers is not possible because the program shows whether the guess was correct, and this is enough to reduce the uncertainty of observers by one.

For the second example, suppose public observation is algorithm complexity of programs. For this purpose, we can use any standard annotation tools which label program's locations with complexity information. CodeHawk [1] is a static analyzer which can automatically label each point of program with complexity information. For this example, we use LeakWatch [Cho14], a tool for estimating information leakage from Java programs. It has two APIs: LeakWatchAPI.secret() tells what are secrets and LeakWatchAPI.observe() tells what are publicly observable. Suppose we have four different complexity levels and each of them is corresponding to a two bits binary as follow: O(c) $\rightarrow$ 00, O(n) $\rightarrow$ 01, O(n*log(n)) $\rightarrow$ 10, and O(n^2) $\rightarrow$ 11.

Example 3. *See Figure 3 that indicates a program annotated with complexity labels (observation) and a 2-bit secret variable (z). Now, we can model the program using transducer which is shown in Figure 4. Then, we can verify k-valued property on transducer model. Given B1 (as public input) is 1 (true), we always observe the same output {01}. Given B1 is 0 (false), we observe two different outputs {10,11}. So, transducer is 2-valued, and the program is not completely confidential. For example, given B1 is zero, observer can guess the range of value for z especially whether the value of z is between 0 and 1, or between 2 and 3 .*

As another example, consider side channel attacks on memory cache [Doy15]. Suppose we have a tool which can show traces (sequence) of hit and miss in cache. In addition, suppose that the data inside the array A[] is secret and we bring each element of array (like A[i]) to cache whenever it is necessary (instead of having all A[] in cache) and keep it in the cache till termination. In this example, let show each hit with 1 and each miss with 0.

[1] `https://github.com/mrbkt/codehawk`

```
void BubbleSort(int a[], int n)
{
    int i, j;
    for (i = 0 ; i < n - 1 ; ++i)
    {
        for (j = 0 ; j < n - 1 - i ; ++j)
        {
            if (a[j] > a[j+1])
            {
                swap(a[j],a[j+1]);
            }
        }
    }
}
```

(a) An implementation of the BubbleSort algorithm

```
void selectionSort(int a[], int n)
{
    int i, j, min;
    for (i = 0 ; i < n - 1 ; ++i)
    {
        min = i;
        for (j = i+1 ; j < n ; ++j)
        {
            if (a[min] > a[j])
            {
                min = j;
            }
        }
        swap(a[j],a[min]);
    }
}
```

(b) An implementation of the SelectionSort algorithm

Figure 5: Two simple sort algorithms

Example 4. *Consider two sort algorithms in Figure 5. We want to show how our model and verification can help compare their confidentiality. For Figure 5.(a) (BubbleSort), if A=[0,3,2,1], we will observe {00,10,11,10,11} as sequence of hit/miss operations in the first iteration (suppose for condition check and swap operation we need two cache access requests). In this observation, {00} means two times miss which occur when we compare A[0] and A[1] for the first time. Since the condition does not hold, we do not have any cache request for swap. In case of third pair 11, we have two times hit that occur when the condition evaluates to true {second pair, 10}, and we have a swap operation for A[1] and A[2]. We can model this sort algorithm in transducer, and if we verify k-valuedness, we will find that the model is 24-valued which means that for each arbitrary order of four elements in A[] (or, for all 24 permutations of 0,1,2,3), we will observe 24-different outputs. By this analysis, we can figure out the order of data initially stored in A[]. For Figure 5.(b) (SelectionSort), if A=[0,3,2,1], we will observe {00,10,10,11} in the first iteration of the algorithm. We can also model this sort algorithm in transducer, and we will find that the algorithm is 1-valued. It means that for all 24 permutations of 0,1,2,3; we will observe the same output. So, observers deduce nothing about the initial order of elements in A[] in case of SelectionSort.*

In the next section, we will present our verification method named ConCert to check k-valued property in abstract model of programs specified in transducer using Z3 theorem prover.

3. Verification of K-valued property

Figure 6 shows the architecture of our tool called ConCert and partial screenshots of each component in the tool. We use three layer architecture to implement ConCert. Given transducer specification in XML file (as abstract model of a program) and user-specified property (using GUI interface), a parser program (written in C++ with Qt Plugin) generates a set of constant definitions, function definitions, and assertion formulae in Z3. Then, they are given to Z3 SMT Solver (using Z3 API in C++), and it will decide whether the property is satisfied in the transducer model using incremental model checking (Note: in this report, we ignore technical proofs which show how we reduce k-valued property in transducers to a set of SMT formulae).

Basically, our verification method depends on notion of counter-example. For example, to check functionality, we should not find a counter-example which exists if there are two different runs such that both of them have the same input, but they produce different outputs. So, the first step to check k-valuedness is to check whether there exists k+1-different runs such that all of them have the same input. For this aim, we need to produce k+1-parallel runs, and we simply use the notion of transducer production to itself k-times. The tool will automatically calculate transducer productions and generate a new XML input file which will be the main input for the rest of verification steps.

The next step is to make sure that k+1 runs under investigation are connected configurations from initial state to final state. This part can be checked using input language over transitions, and we rely on flow relations and Parikh image definition [Dem10]. Consider q_0 and q_f as initial and final states respectfully without direct transition from q_f to q_0. A path is a finite sequence of transitions ($\pi = t_1,...,t_k$) such that for i $\in$ [1,k-1], end(t_i) = beg(t_{i+1}). The *image* of $\pi = t_1,...,t_k$ is a map I: $\delta \rightarrow \mathcal{N}$ that counts how many times each transition is used in π shown as $\vec{image}=(x_1,...,x_k)$ (a vector of all transition frequency in π). For every state q' in the set of all states except initial and final states (Q \ $\{q_0,q_f\}$), the number of transitions entering

in q' is equal to the number of transitions going out of q'. The number of transitions entering in q_0 is one less than the number of transitions going out of q_0. Similarly, the number of transitions entering in q_f is one more than the number of transitions going out of q_f. In addition, frequencies of all other transitions (which are not part of the path) should be equal to zero.

So far, we have checked the existence of k+1 connected and successful runs such that all of them have the same input. Otherwise, the tool returns UNSAT which means that k-valued property is satisfied. The next step is to find k+1 different output symbols in the same evaluation of counters. To show that our verification method is decidable, we prove that checking k-valued property can be reduced to 1-reversal bounded multi-counter machine [Iba78] which is decidable. For the sake of simplicity and without losing generality, let's discuss how we check counter-example for 1-valued (functionality). We simulate two runs and consider two counters which increase by one for each output symbol in each run. Then, we non-deterministically guess a counter value and capture the output symbol under that value of counter. We find counter-example for 1-valued whenever we can find two counter values such that their values are the same, but the output symbols under the values are different in two successful runs.

Considering these concepts, let see how ConCert checks k-valued property for the example of Figure 1.(a). This transducer is given to the program in the form of XML file and 1-valued is specified as input property by user (Figure 6 shows working flow for this example). First, the tool generates XML file which specifies square of this transducer. Then, the parser program will automatically generate SMT formulae to check 1-valuedness from the square transducer model. In the first step of formula generation, the program generates all necessary constant definitions. In the next step, the tool generates functions like transition functions Ti which uses counter functions Cij as follow (i is run index and j is transition index):

```
(*): (define-fun T0 ((k Int)(o Int)(index Int)) Bool
     (ite (or (and (= k 1)(= (C01 o index) true))
     (and (= k 2)(= (C02 o index) true))
     ...

(*): (define-fun C01 ((s Int)(ind Int)) Bool
     (ite (or (and (= s -1)(= ind 0))(and (= s 0)(= ind 1))(and (= s 1)(= ind 2))) true false))
     ...
```

These definitions formulate the transitions with their corresponding index number (defined as variable k for each transition), the possible value of counter for given transition, and the corresponding output symbol for that transition in the first run (for second run, we will have T1 and C1j definitions). In the next step, ConCert generates assertion formulae (the tool generates 52 SMT formula assertions for this example, and we will show some of them here). The following assertion is one of the path formulae.

```
(*): (assert (and (= (+ (* -1 x1) (* -1 x2) (* -1 x3) (* -1 x4) ) -1)(= (+ x21 0) 1)))
```

This formula shows flow relations for one node in which (x1,x2,x3,x4) are incoming transitions and (x21) is only outgoing transition. Then, we define variables to show counter values in each run (D_0 is for the first run and D_1 is for the second run) based on counter variables Yij for each transition (i is run index and j is transition index):

```
(*): (assert (= D0 (+ Y01 Y02 ...  Y021)))

(*): (assert (= D1 (+ Y11 Y12 ...  Y121)))
```

Now, we need assertions for all possible guesses of output symbols and their corresponding counter values. The tool generates assertions for each transition to include all possible guesses of output symbols. For this example, we will have 24 assertions of this type, and one of them is given as follow:

```
(*): (assert (or (and (> x1 0)(<= Y01 2)(>= Y01 1)(= (T0 1 o01 Y01) true))(and (= x1 0)(= Y01
     0)(= (T0 1 o01 Y01) true))))
```

It says that if transition x1 is taken, then corresponding counter variable (Y01) can be 1 or 2, and corresponding symbol value (o01) will be determined by calling function T0. Otherwise, Y01 will be zero and the symbol will be -1 (after calling T0). Finally, we have four relations which go through all o0j and o1j (j is transition index) from previous relations to check whether there exist o0 (one sample of output symbols in the first run) and o1 (one sample of output symbols in the second run) such that they are evaluated to two

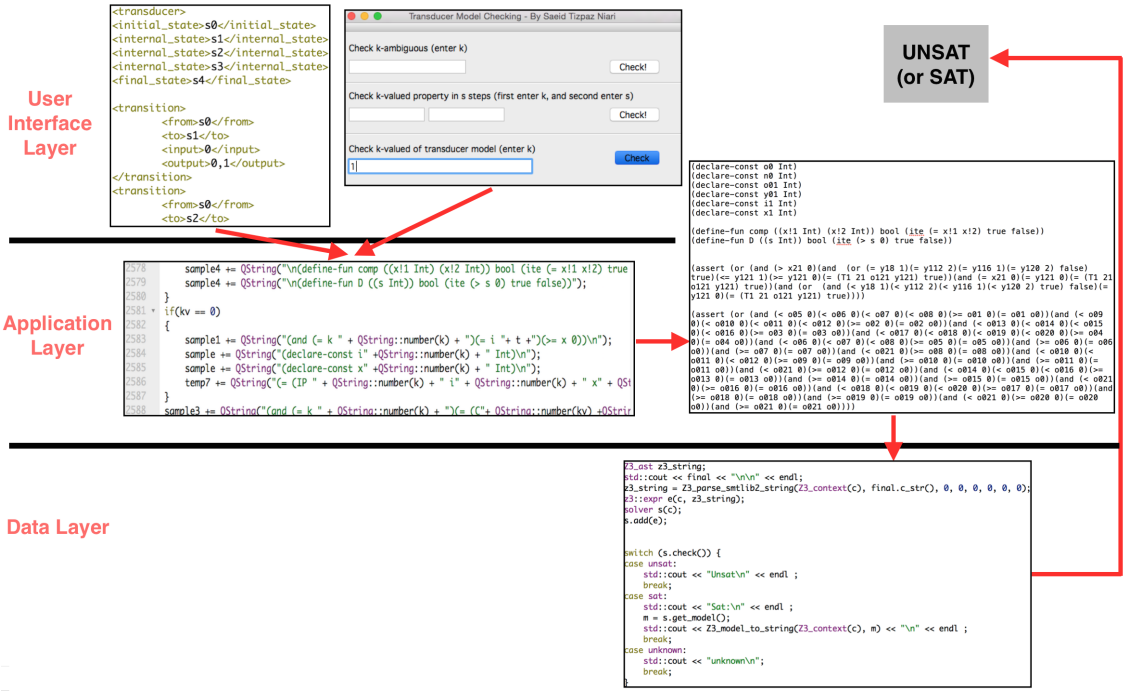

Figure 6: Architecture and working flow for ConCert

different symbols (o0 ≠ o1) while they are under the same value of counters (D0 = D1). If the formulae are satisfied, the transducer is not functional, and we will check 2-valuedness in the next step. Otherwise, it is functional and holds non-interference. For this example, the tool will return UNSAT which means that the transducer is functional.

4. Experimental Result

Table 1 shows experimental results obtained from running ConCert for different models with different scales. Note that maximum length of assertion (Max Length. Assert) shows the biggest assertion formula based on the number of clauses in the formula.

Table 1: Experimental results from verification of different models

Name	Num. State	Num. Transition	Property	Num. Assert	Max Length. Assert	Time	SAT?
Model1	5	7	1-valued	52	21	70ms	SAT
Model2	11	16	1-valued	358	174	2s	UNSAT
Model3	20	32	1-valued	1702	846	40s	UNSAT
Model4	25	40	1-valued	2758	1374	120s	UNSAT

5. References

[Geo09] Doychev, Goran, et al. CacheAudit: a tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security*, 18.1 (2015): 4.

[Doy15] Smith, Geoffrey. On the foundations of quantitative information flow. *Foundations of software science and computational structures* Springer Berlin Heidelberg, 2009. 288-302.

[Dem10] S. Demri, Decidable Properties for Counter Systems *Advanced notes on Counter System,* ESSLLI 2010, Copenhagen, Denmark, August 2010

[Cho14] Chothia, Tom, Chothia, Tom, Yusuke Kawamoto, and Chris Novakovic. LeakWatch: Estimating information leakage from java programs. *Computer Security-ESORICS,* Springer International Publishing, 2014. 219-236.

[Iba78] Ibarra, Oscar H, Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM (JACM),* 25.1 (1978): 116-133.