1 Introduction

Compiler transformations can be correct yet insecure. Here, we have proposed a translation validation approach for verifying the security of compiler transformations with respect to information flow.

1.1 Information Leakage

Variables can be partitioned into high security (H) and low security (L). All state variables are considered to be low security and the input variables can be of high or low securities. Let we have three values, namely a, b and c, where a and b are the values of a high variable such that $a \neq b$ and c is a low variable value. There are two input pairs (a, c) and (b, c). If for a program S, the computation of S on both the input pairs either differs in the sequence of output values or the value of one of the low variables differ at their final states when both terminates, then program S is said to leak information and (a, b, c) is called as a leaky triple for program S.

1.2 Correct Transformation

Program transformations does not alter the set of input variables. A transformation from program S to program T may alter the code of S or the set of state variables. The transformation is correct if, for every input value a, the sequence of output values for executions of S and T from the input value a is identical.

1.3 Secure Transformation

A transformed program T after the register allocation from a source program S is said to be secure, if all the leaky triples which belong to the transformed program, also belong to the source program. It ensures relative security, i.e. the transformed program T is not more leaky than the source program S.

Suppose that the transformation from to S to T is correct. Consider a leaky triple (a, b, c) for T. If the computations of T from inputs (H = a, L = c) and (H = b, L = c) differ in their output, from correctness, this difference must also appear in the corresponding computations in S. Hence, the only way in which T can be less secure than S is if both computations terminate in T with different values for low-variables, while the corresponding computations in S terminate with identical values for low-variables.

2 Program Model

Any program can be represented as an finite state machines with datapaths (FSMDs). An FSMD is a universal specification model that can represent a program efficiently. The translation validation of information leakage is formulated on FSMDs in this work. However, the

same formulation can be adapted on other program models like CDFG as well. The FSMD is defined as an ordered tuple $\langle Q, q_0, I, V, O, f, h\rangle$ where

1. $Q = \{q_0, q_1, q_2, \cdots, q_n\}$ is the finite set of control states,
2. $q_0 \in Q$ is the reset state,
3. I is the set of primary input signals where $I = I_h \cup I_l$, where I_h is the set of high inputs and I_l is the set of low inputs,
4. V is the set of storage variables, and Σ is the set of all data storage states or simply data states,
5. O is the set of primary output signals,
6. $f : Q \times 2^S \rightarrow Q$ is the state transition function,
7. $h : Q \times 2^S \rightarrow U$ is the update function of the output and the storage variables, where S and U are defined as follows.
   (a) $S = \{L \cup E\}$ is the set of status expressions, where L is the set of Boolean literals of the form b or $\neg\, b$, $b \in B \subseteq V$ is a Boolean variable and E is the set of arithmetic predicates over $I \cup (V - B)$. Any arithmetic predicate is of the form eR0, where e is an arithmetic expression and $R \in \{==, \neq, >, \geq, <, \leq\}$.
   (b) U is a set of storage or output assignments of the form $\{$ x $\Leftarrow$ e|x $\in$ O $\cup$ V , and e is an arithmetic predicate or expression over I $\cup$ (V $-$ B) $\}$ that represents a set of storage or output assignments.

A *path* from q_i to q_j is a sequence of state transitions of the form $\langle q_i \xrightarrow[c_i]{} q_{i+1} \xrightarrow[c_{i+1}]{} \cdots \xrightarrow[c_{i+n-1}]{} q_{i+n} = q_j\rangle$ where $q_k \in Q$ for all k, $i \leq k \leq i+n$, and $\exists c_k \in 2^S$ such that $f_k(q_k, c_k) = q_{k+1}$ for all k, $i \leq k \leq i+n-1$. A *(finite) path* α is where all the states are different, except the end state q_j may be the same as the start state q_i.

The *condition of execution* R_α of a path α is a logical expression over $I \cup V$, which must be satisfied by initial data state in order to traverse the path α.

The *data transformation* r_α of a path α is an ordered pair $\langle s_\alpha, O_\alpha\rangle$, where s_α is an updated variable vector and O_α is an updated output list after executing α. Thus R_α is the weakest precondition of the path α with respect to r_α.

For a path α, R_α and r_α are computed by forward or backward substitution based on symbolic execution.

A *computation* or *trace* of an FSMD is a path or a concatenation of two or more paths from the reset state q_0 to itself and q_0 should not occur in between. For an FSMD M, any computation τ is the concatenation $[\alpha_1\alpha_2\cdots\alpha_n]$ of paths of M where for all k, $1 \leq k < n$, α_k terminates in the start state of the path α_{k+1}, q_0 is the start state of α_1 and the end state of α_n.

Two paths β and α are equivalent (U-equivalent), denoted by $\beta \simeq \alpha$ if $R_\beta \equiv R_\alpha$ and $r_\beta = r_\alpha$. The equivalence of two computations or traces can be defined in a similar fashion.

Definition 2.1 (Path Cover of an FSMD)**.** *A finite set of paths $P = \{p_0, p_1, \ldots, p_k\}$ is said to be a path cover of an FSMD S if any computation τ of S can be looked upon as a concatenation of paths from P.*

To obtain a path cover, FSMD breaks down into smaller segments by introducing cutpoints so that each loop in an FSMD is cut in at least one cutpoint. This is based on the Floyd–Hoare method of program verification []. The set of all paths from a cutpoint to another cutpoint without any intermediary occurrences of cutpoint is a path cover of the FSMD. We denote the path cover of S and T as P_s and P_t, respectively.

3 Problem Statement

The source program S is transformed into T by applying a set of compiler optimizations. Is T as secure as S with respect to information leakage?

3.1 Objective

To show that for every leaky triple of T, there exist an equivalent leaky triple in S or if there is a leak in T there must be a corresponding leak in S.

Since the possibility of leaky triple may be infinite, identifying all leaky triple is not possible. Instead we aim for symbolic execution (specifically symbolic bi-simulation) based approach.

3.2 Our approach

The input program is modelled as the FSMD $S = \langle Q_s, q_s, I, O, V_s, f_s, h_s \rangle$ and the transformed program is modelled as FSMD $T = \langle Q_t, q_t, I, O, V_t, f_t, h_t \rangle$. Both S and T have identical input(s)/output(s). However, their internal variables may differ due to compiler optimizations. Therefore, V_s may not equal to V_t. Moreover, compiler optimizations may change the operations and the control structure as well. Therefore, the state transition functions and the update function for S and T may not be identical as well. The inputs of S and T consist of a set of high variables and a set of low variables. Therefore, $I = I_h \cup I_l$.

The following theorem captures the relative leakiness between two behaviours.

Theorem 3.1. *An FSMD T is said to be as leaky as S, i.e., $T \simeq_R S$, iff for each trace τ_t in T, there exist a corresponding trace τ_s in S such that $\tau_t \simeq_R \tau_s$. Here $\tau_t \simeq_R \tau_s$ denotes that τ_t is as leaky as τ_s.*

Proof. Let for each trace τ_t in T, there exists a corresponding trace τ_s in S such that $\tau_t \simeq_R \tau_s$. Consider a leaky triple (a, b, c) for trace τ_t in T. Let the induced computations on the inputs (H = a,L = c) and (H = b,L = c) be τ_{ta} and τ_{tb}. Let the final states of τ_{ta} and τ_{tb} is q_{ta_f}

and q_{tb_f}. As the leak is in trace τ_t, so there is a mismatch of values for some low variable in q_{ta_f} and q_{tb_f}. From the above hypothesis, it follows that there exists a corresponding trace τ_{sa} and τ_{sb} in S whose final states q_{sa_f} and q_{sb_f} related to q_{ta_f} and q_{tb_f} respectively. So there must be a mismatch of values for some low variable in q_{sa_f} and q_{sb_f}. This leads to a leak in τ_S which implies T is as leaky as S. □

Based on the above theorem, following method can be used to check the relative information leakage between two behaviours S and T.

1. Find all the traces in both S and T.
2. Find all corresponding traces between S and T (either by equivalence checking or by data driven approach)
3. Both S and T has same high input variables. So, T can not have a leak of a high variable which is not present in S. For each corresponding trace pair (τ_s, τ_t), compute data transformation s_{τ_s} in S and s_{τ_t} in T. Set the variables in S and T as V_s and V_t respectively. Note that V_s may not be equal to V_t. Now, check for the following.
   (a) $\forall v \in V_s \cap V_t$ (common variables in S and T), the high variables involved in corresponding expression in s_{τ_s} and $s_{\tau t}$ must be the same. If there is a high variable in $s_{\tau_t}|_v$ which does not present in $s_{\tau_s}|_v$, it means τ_t is more leaky than τ_s.
   (b) $\forall v \in V_t - V_s$ (new variables in T), if $s_{\tau_t}|_v$ involve any high variable, it means τ_t is leakier than τ_s. In this case, v will leak information in τ_t and v does not present in τ_s.
   (c) $\forall v \in V_s - V_t$, we do not need to check for the v.
   (d) If $\tau_s \simeq \tau_t$, then output sequence must be the same in both τ_s and τ_t. If the equivalence of τ_s and τ_t can not be proved, the output sequence O_{τ_s} and O_{τ_t} must be the same.

Problem with the above approach: Because of unbounded loop, there may be large or infinite number of traces in a program. Therefore, finding all possible traces and checking their correspondence may not be feasible in practice. Like program equivalence checking, we, therefore, insert cut points in the program so that each loop must cut by at least one cut point. This is similar to Floyd-Hoare logic of program verification. The set of paths between cutpoints without having an intermediate cutpoints are the *path cover* of the FSMD. With path cover, theorem 3.1 can be rewritten as

Theorem 3.2. *An FSMD T is said to be as leaky as S, i.e., $T \simeq_R S$ iff for each path p_i in the path cover P_t of T, there exists a path p_j in the path cover P_s of S such that $p_i \simeq_R p_j$. Formally, $T \simeq_R S$ iff $\forall p_i \in P_t, \exists p_j \in P_s$ s.t. $p_i \simeq_R p_j$.*

Proof. $T \simeq_R S$, iff for each trace τ_t in T, there exist a corresponding trace τ_s in S such that $\tau_t \simeq_R \tau_s$ [by Theorem 3.1] . Now, let there exists a finite path cover $P_t = \{p_{t0}, p_{t1}, \cdots, p_{tl}\}$ of T, let a set $P_s = \{p_{s0}, p_{s1}, \cdots, p_{sl}\}$ of S, corresponding to P_s exists such that $p_{ti} \simeq_R p_{si}, 0 \leq i \leq l$. Since P_t covers T, any trace of τ_t of T can be looked upon as a concatenated path $\{p_{ti_0}, p_{ti_1}, \cdots, p_{ti_n}\}$ from P_t. From the above hypothesis, it follows that there exists a sequence μ_x of paths $\{p_{sj_0}, p_{sj_1}, \cdots, p_{sj_n}\}$ of P_s where $p_{ti_k} \simeq_R p_{sj_k}, 0 \leq k \leq n$. Therefore, we need to prove that μ_x is a trace which is a concatenated path from P_s. As τ_t is a trace of T, it must emanates from reset state q_t i.e. p_{ti_0} emanates from q_t. Therefore, the sequence μ_x must emanates from reset state q_s as $p_{ti_0} \simeq_R p_{sj_0}$. Hence, μ_x is a concatenated paths representing a trace τ_s of S, where $\tau_s \simeq_R \tau_t$. □

4 Proposed Approach

In this work, we choose cutpoints in an FSMD as

1. the reset state is a cutpoint
2. Any loop entry point is also a cutpoint

The above chosen cutpoits indeed will cut each loop in a behaviour. Therefore, the set of paths obtained between cutpoints without having any intermediate cutpoints will be a path cover []. We denote the path cover of the FSMDs S and T as P_s and P_t, respectively.

We assume following in our formulation:

1. No loop merging happens due to compiler optimizations, i.e. number of loops in S and T both are same.
2. If conditional transformation takes place inside an if-else block, it will be pre-processed. The number of paths inside an if-else block of S are the same of the corresponding if-else block of T.
3. Compatible if-else may be merged by the compiler. However, it will not change the number of paths in path covers.

Under the above assumptions, P_s and P_t will always have same number of paths.

5 Leak Propagation

Let $I_h = \{I_{h0}, I_{h1}, \cdots, I_{ha}\}$. For a path α, α^s denotes the start state of the path α and α^f denotes the end state of path α. A path $\alpha : \alpha^s \Rightarrow \alpha^f$ is associated with an order tuple $\gamma_\alpha = \langle n_1, n_2, \cdots, n_k \rangle$ and n_i is an integer representing the leak propagation through variable v_i, $\forall v_i \in V_s \cup V_t$. Specifically, n_i is a x bit number where j^{th} bit in n_i represents the leak of high input I_{hj} through variable v_i in the path α. $n_i = \langle b_{ix-1}, b_{ix-2}, \cdots, b_{i1}, b_{i0} \rangle$, where $b_{ij} = 1$ if the variable v_i leaks high input I_{hj} in the path α; $b_{ij} = 0$, otherwise.

5.1 How to compute Propagated Leak

For the path α, the $r_\alpha = \langle s_\alpha, O_\alpha \rangle$, where $s_\alpha = \langle e_1, e_2, \cdots e_k \rangle$, where e_i represents the final value of the variable v_i at the end state α^f of α. Here k is the number of variables in $V_s \cup V_t$. Here, e_i is an algebraic expression over $I \cup V_s \cup V_t$. If the high input variable I_{hj} presents in e_i, then b_{ij} is set to 1 in n_i corresponding to variable v_i. By examining all variables present in e_i, all the bits of n_i is set/reset. In case of a propagated leak in the state α^s, say $\gamma_{\alpha_s} = \langle \vec{n} \rangle$ where n represents the vector of k integers, the propagated leak of α is computed as $\gamma_\alpha = \gamma_\alpha \vee \gamma_{\alpha_s}$ i.e., bit wise OR of the corresponding integers in γ_α and γ_{α_s}. If initially $\gamma_\alpha = \langle n_1, n_2, \cdots, n_k \rangle$ and $\gamma_{\alpha_s} = \langle n_1^{'}, n_2^{'}, \cdots, n_k^{'} \rangle$, then after leak propagation $\gamma_\alpha = \gamma_\alpha \vee \gamma_{\alpha_s} = \langle n_1 \vee n_1^{'}, n_2 \vee n_2^{'}, \cdots, n_k \vee n_k^{'} \rangle$. Intuitively, new leak is the union of the propagated leak and leak of the current path. The propagated leak of α, i.e. γ_α, be also referred as the leak γ_{α^f} associated with the final state α^f of α.

5.2 Relative Leakiness of Paths

Definition 5.1 (Relative Leakiness of Paths)**.** *A path $\beta : \beta^s \Rightarrow \beta^f$ of T with propagated leak γ_{β^s} is said to be as leaky as a path $\alpha : \alpha^s \Rightarrow \alpha^f$ of S with propagated leak γ_{α^s} iff $\gamma_\beta \simeq \gamma_\alpha$ i.e. if $\gamma_\beta = \langle n_{\beta_1}, \cdots, n_{\beta_k} \rangle$ and $\gamma_\alpha = \langle n_{\alpha_1}, \cdots, n_{\alpha_k} \rangle$, then $n_{\beta_i} = n_{\alpha_i}, \forall i = 1, 2, \cdots, k$. Then β is as leaky as α is denoted as $\beta \simeq_R \alpha$.*

Otherwise, the path α is said to be conditionally as leaky as β, denoted as $\alpha \simeq_C \beta$.

Definition 5.2 (Conditional Leakiness of Paths)**.** *1. $\alpha^f \neq q_s$ and $\beta^f \neq q_t$*

2. For all path $\beta^{'}$ emanating from β^f with propagated leak $\gamma_{\beta f}$ in T, $\exists \alpha^{'}$ from α^f with propagated leak γ_{α^f} in S such that $\beta^{'} \simeq_R \alpha^{'}$ or $\beta^{'} \simeq_C \alpha^{'}$

Condition 1 prevents leak propagation beyond reset state. It also ensures that if S and T are equally leaky, then $\alpha \simeq_R \beta$ must hold when $\alpha^f = q_s$ and $\beta^f = q_t$

The final states of two conditionally or equally leaky paths are denoted as corresponding states.

Definition 5.3 (Corresponding States)**.** *Two FSMDs $S = \langle Q_s, q_s, I, O, V_s, f_s, h_s \rangle$ and $T = \langle Q_t, q_t, I, O, V_t, f_t, h_t \rangle$ having identical inputs/outputs, the correspondence of states between S and T are defined as follows:*

1. *The reset states q_s and q_t are corresponding states, i.e., $\langle q_s, q_t \rangle$.*
2. *The states $q_{sm} \in Q_s$ and $q_{tn} \in Q_t$ are corresponding states, i.e., $\langle q_{sm}, q_{tn} \rangle$ if the state $q_{si} \in Q_s$ and $q_{tj} \in Q_t$ are corresponding states, i.e., $\langle q_{si}, q_{tj} \rangle$ and there exists paths, $\alpha : q_{si} \Rightarrow q_{sm}$ and $\beta : q_{tj} \Rightarrow q_{tn}$, such that $\alpha \simeq_R \beta$.*

5.3 Leak Propagation Over Loop

Each path of the loop will be traversed once. In Fig. 1, the propagated leak at state q_y i.e. $\gamma_{q_y} = \gamma_{q_y} \vee \gamma_\alpha$. Basically, it is the OR of the propagated leak α and the leak of the loop.

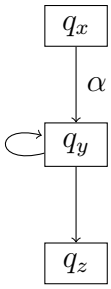

Figure 1: Leak Propagation Over Loop

5.4 Correspondence Paths in Path Cover

We can use data driven approach to find the correspondence of paths in P_s and P_t. The basic idea here is to run random tests on both program, then compare the value of common variables in corresponding states pair and conclude the correspondence of paths based on maximum similarity. Corresponding states pair are obvious or easy to find because we assume that no loop merging happens and only loop entry or exit points are cutpoints. Lets assume the correspondence of paths is obtained as $\langle P_{si}, P_{ti} \rangle$, $\forall i = 0, 1, \cdots, l$, $P_{si} \in P_s$ and $P_{ti} \in P_t$.

5.5 Translation Validation Algorithm

Algorithm 1 Translation_Validation(S, T)

1: Compute cutpoints and path cover P_s and P_t in S and T
2: Find correspondence of cutpoints and paths in P_s and P_t
3: For all paths in P_s and P_t, set the default propagated leak as vector of 0
4: CSP is the set of corresponding state pairs, ER is the set of Relationally Equivalent state pairs and EC is the set of Conditionally Equivalent state pairs
5: **for** each (q_{si} and q_{tj}) in CSP **do**
6: Check_Correspondence($q_{si}, q_{tj}, \gamma_{\alpha f}, \gamma_{\beta f}, ER, EC$)
7: **end for**

Algorithm 2 Check_Correspondence($q_{si}, q_{tj}, \gamma_{\alpha^f}, \gamma_{\beta^f}, ER, EC$)

1: **for** each path β in P_t emanating from q_{tj} **do**
2: find α = corresponding path in P_s emanating from q_{si}
3: Compute $R_\alpha, r_\alpha, R_\beta, r_\beta$
4: Compute the propagated leak γ_α and γ_β with respect to the $\gamma_{q_{si}}$ and $\gamma_{q_{tj}}$ respectively
5: **if** $\alpha \simeq_R \beta$ **then**
6: $ER = ER \cup (\alpha, \beta)$
7: **else**
8: **if** $\alpha^f = q_s$ and $\beta^f = q_t$ **then**
9: T is leakier than S and report the leaky variables
10: **else**
11: Compute the difference of γ_α and γ_β
12: $\gamma_{\alpha^f} = \gamma_\alpha - \gamma_\beta$
13: $\gamma_{\beta^f} = \gamma_\beta - \gamma_\alpha$
14: $EC = EC \cup (\alpha, \beta)$
15: Check_Correspondence($q_{si}, q_{tj}, \gamma_{\alpha^f}, \gamma_{\beta^f}, ER, EC$)
16: **end if**
17: **end if**
18: **end for**