Reducing False-positive and False-negative Warnings in Static Taint Analysis

Anonymous Authour(s)

I. Background

Taint analysis is widely used in security tasks. In industrial scenarios, on one hand, the taint analysis tools report false-negative warnings due to the complexity of product codes. For this challenge, we focus on the specific scenario that taint analysis is performed by specifying members of struct variable as taints. As an instance, the struct variable may change its type by calling cast statement where the taint mark of the struct members would be lost. On the other hand, the taint analysis tools have serious false-positive problem. For this challenge, we focus on (1) buffer size-aware scenarios and (2) path constraint-aware scenarios. We wonder whether there exists efficient approaches of calculating buffer size and path constraints to reduce false-positive warnings.

II. Challenge

This challenge is about automating efficient and effective taint analysis for industry-scale codes. It targets at the following specific scenarios.

Case 1: Precise propagation of taint marked on struct members: How can we optimize the propagation of struct member taints originally marked by the user, in the process of interprocedural data flow where over-approximation of type cast statements exist (including conversion to char *, void *, or other types of structures, etc.). An example is shown in Fig 1.

Case 2&3: Buffer size calculation: Program statements such as for-loop (case 2) statements or type-cast (case 3) statements seriously restrict the calculation of buffer size. Example are shown in Fig 2, 3.

Case 4: Path Constraint Calculation: Calculating path constraints still lacks an efficient and general solution. An example is shown in Fig 4.

Targeted Language: C/C++.

III. Deliverables

Based on LLVM IR analysis, a **solution and demo** should be provided which supports taint analysis to address at least one of the aforementioned four cases. The solution will not cause any performance bottleneck in actual projects, i.e., the overhead should be less than 15%.

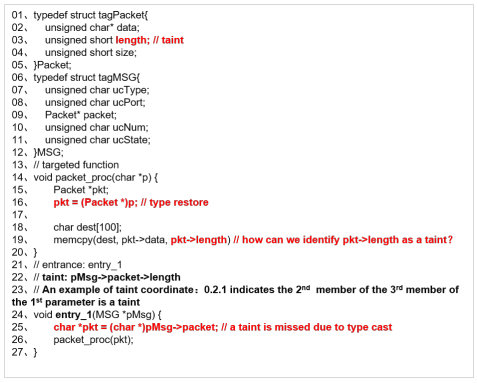

Fig. 1. An example of false-negative warning caused by type cast of struct variable: on line 16, the variable **p** is cast from char * to Packet, while on line 25, a member called **packet** of the struct variable is cast from Packet to char *. In these cases, the taint is lost during the propagation.

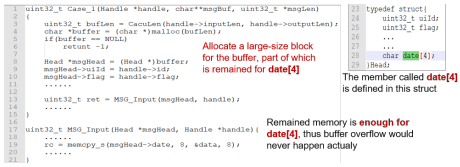

Fig. 2. An example of false-positive warning caused by type cast of struct variable: on line 19, the size of the 1st parameter is 4 while the size of the 2nd parameter is 8, thus the static analysis tool reports a warning, which is actually false positive since on line 4, enough memory is allocated for **date[4]**.

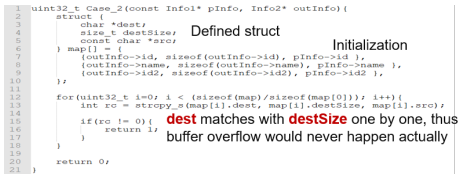

Fig. 3. An example of false-positive warning due to over-approximation of loop statement: on line 13, the size of the 1st parameter is 12 while the size of the 2nd parameter is 28, thus the static analysis tool reports a warning, which is actually false positive since on line 13, **dest** matches with **destSize** one by one.

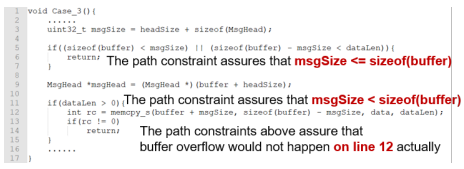

Fig. 4. An example of false-positive warning due to over-approximation of path constraints: on line 12, the static analysis tool reports a warning, which is actually false positive if path constraints are considered.