Static Information Flow Tracking (SIFT) Analysis for Hardware Design Verification

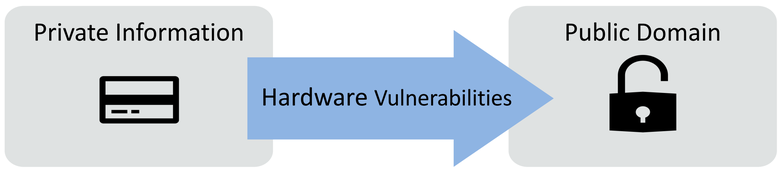
Christie Lincoln, Lisa Luo, Amir Uqdah, Alvin Zhang
Advisors: Armaiti Ardeshiricham, Ryan Kastner

ERSP UC San Diego Early Research Scholars Program

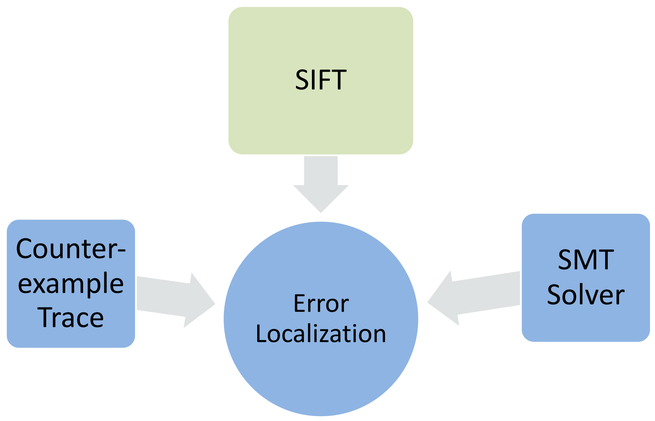
UCSDCSE Computer Science and Engineering

Motivation

Recently discovered attacks that exploit vulnerabilities in popular hardware allow private information to be leaked to public domains.



To prevent these attacks, hardware designers must eliminate these vulnerabilities. One approach is by using verification to uncover such vulnerabilities and error localization to resolve them.



Error localization can identify where a program is failing a security property. A key aspect of this is the need to track inputs through a system. One way is through information flow tracking.

Background and Problem

Information Flow Tracking (IFT) can be used to identify what inputs affect which outputs. This is helpful in verification to determine if private inputs flow to public outputs, which indicates an information leak.

Previous IFT Tools

Gate Level IFT (GLIFT) — Large overhead and lower level abstraction makes this tool less effective in verification.

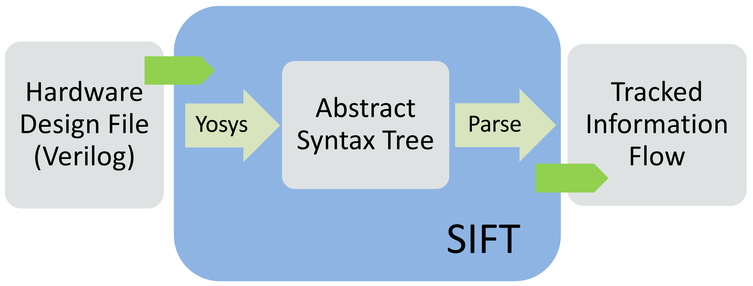
Register Transfer Level IFT (RTLIFT) — Attempts to solve the issues with GLIFT but only returns binary answers as to whether or not an input affects an output.

We want to develop a tool that provides more information, such as the source of the leakage and the path of the leaked data.

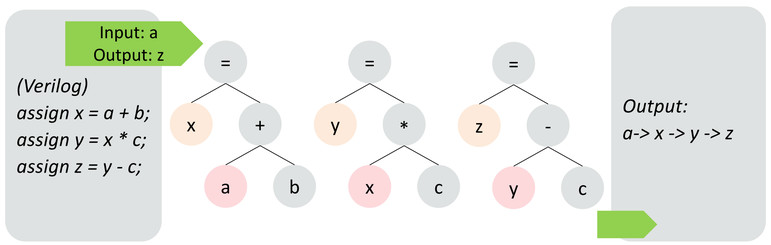
Proposed Solution — Trace the path of inputs through the system to discover exactly when private information flows to public domain.

Our Approach: Static IFT (SIFT)

We analyze Verilog code by using a framework called Yosys, which creates an Abstract Syntax Tree (AST) of the design, and analyzing that resulting AST. We label sensitive input variables as tainted and as they interact with other variables, they taint output variables.



In the following example, we mark `a` as tainted. `a` taints `x`, which taints `y`, which taints `z`.



We print out all the variables that the tainted inputs taint. Based on user input, we either print the tainted paths of each output or the tainted path of a specific output.

Novelty — Our tool identifies this path whereas previous tools would only determine that `a` affects `z`.

Example — If `a` and `x` were both private, but `y` and `z` were both public, the first instance of the leak actually occurred in the second statement.

Impact — Our tool helps hardware designers easily minimize security flaws by identifying where the leakage precisely occurs.

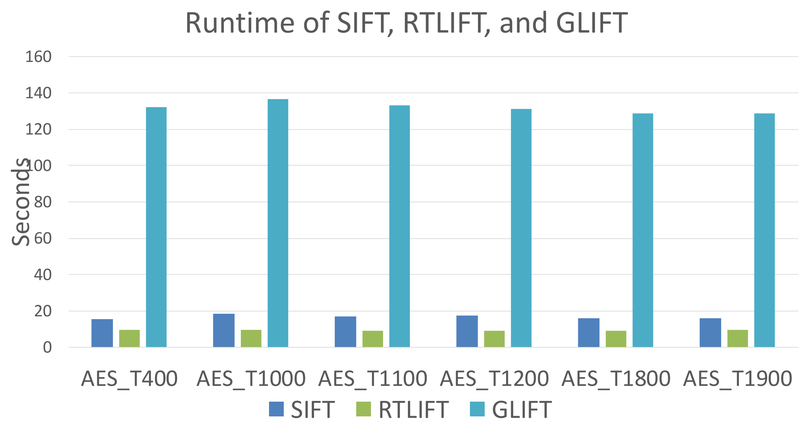
SIFT's Impact to Error Localization:

SIFT — SIFT is given all inputs that are critical to a failing security property of the system

SMT Solver — The State variables that SIFT identifies are are tested to determine if they are critical to the verification failure

Results

After testing simple, singular modules simulating simple basic arithmetic methods, we tested more complex programs, which we gathered from Trust-Hub, a hardware security resource funded by the National Science Foundation (NSF). These programs contained current hardware encryption algorithms that had certain security design flaws.



Based on our tests, our tool, SIFT, is slower than RTLIFT but faster than GLIFT. However, SIFT provides more information than RTLIFT because SIFT provides the path of information leakage.

Ongoing: Error Localization

Verification — Run program with security properties through SMT solver to generate counterexample

Counterexample Reduction — Use SMT solver to determine which inputs are critical to failing the property

SIFT — Use SIFT to find which state variables are affected by critical inputs

Compute suspicious state variables — Use SMT solver to determine which state variables returned by SIFT are critical to failing security properties

We have preprocessed and tested various benchmarks in order to ensure the validity of both SIFT and the error localization approach as a whole. Our next step is to expand evaluation by synthesizing benchmarks that are more relevant in the realm of verification

Acknowledgements

We would first like to thank Professor Ryan Kastner and Armaiti Ardeshiricham. Their help and guidance has been invaluable throughout this process. We would also like to thank Professor Christine Alvarado for giving us this opportunity to work on this project through the Early Research Scholars Program. (ERSP). ERSP is a program at UC San Diego aimed at introducing undergraduate students to graduate level research.