Sparse Representation of Implicit Flows with Applications to Side-Channel Detection

• Bruno Rodrigues
• Diego Aranha
• Fernando Pereira

UNICAMP
Goal

The goal of this work is to implement information flow analysis precisely and efficiently in low level languages.
Goal

The goal of this work is to implement information flow analysis precisely and efficiently in low level languages.

- Information flows from variable $s$ to variable $d$ if the value of $s$ determines the value of $d$.
  - Explicit flows: $d = f(..., s, ...)$
  - Implicit flows: $d = s ? 0 : 1$
Contributions

The goal of this work is to implement information flow analysis precisely and efficiently in low level languages

• We avoid worst cases in the technique used by Ferrante et al to build Dependence Graphs†.

†: The program dependence graph and its use in optimization – TOPLAS – 1987
Contributions

The goal of this work is to implement information flow analysis precisely and efficiently in low level languages

• We avoid worst cases in the technique used by Ferrante et al to build Dependence Graphs.
• We reduce the complexity of implementing Hunt and Sands' System of Security Types†.

†: On flow-sensitive security types – POPL – 2006
Contributions

The goal of this work is to implement information flow analysis precisely and efficiently in low level languages.

- We avoid worst cases in the technique used by Ferrante et al to build Dependence Graphs.
- We reduce the complexity of implementing Hunt and Sands' System of Security Types.
- We have detected security issues in real-world crypto libraries, such as TrueCrypt and OpenSSL.
The Challenge

- Find an efficient way to track implicit flows.

```c
int b;
if (p) {
    b = 0;
} else {
    b = 1;
}
```

- This code contains an implicit flow of information from predicate `p` to variable `b`.
The Key Idea

"The paper proposes an analysis to efficiently discover and represent implicit information flows in programs in static single assignment (SSA) form. It achieves this by focusing on control dependencies from values to values instead of from values to instructions"\(^\dagger\).

- Static Single Assignment $\rightarrow$ sparse analysis
- Control dependences from values to values

\(^\dagger\): quoting comment from anonymous referee (CC'16)
Dependence Graphs – Original View†

- Vertices are instructions
- Edge from $i_0$ to $i_1$ if $i_0$ may determine the behavior of $i_1$

Dependence Graphs – Our View

- Vertices are variables in the SSA-form program
- Edge from $v_0$ to $v_1$ if the value of $v_0$ determines the value of $v_1$

†: The program dependence graph and its use in optimization – TOPLAS – 1987
int genKeyMask(
    int seed,
    int* t1,
    int* t2
) {
    uint_64 r;
    int i = 0;
    r = random(seed);
    while (i < 64) {
        if (r & 1) {
            t1[i] = 1;
        } else {
            t2[i] = 1;
        }
        r >>= 1;
        i++;
    }
}
An Algorithm to Find Implicit Flows

This is our algorithm, in SML/NJ's syntax

You do not have to understand it now.

– But it is pretty short
– And goes down the programs' dominance tree
– And needs also the program's post-dominance tree
Important: Dominance

- Node 'a' dominates node 'b', because any path from • to 'b' goes across 'a'. Similarly, node 'b' post-dominates node 'a', because any (backwards) path from ◐ to 'a' must go across 'b'.
- Node n post-dominates node m, but m does not dominate n.
- Node x dominates node y, but y does not post-dominate node x.
We traverse the dominator tree, "remembering" the predicates that we find on the way. For instance, after walking on the red path, we will have visited predicates $b_0$, $b_1$ and $b_2$. 
We "forget" a predicate p if (i) we are done visiting every child of b, the block where p is used in a branch, or (ii) we reach the basic block d that post-dominates b.
We start visiting the first basic block, which we shall call $b_{0-2}$. At the beginning, we have seen no predicate. This block, $b_{0-2}$, terminates with an unconditional jump; hence, we do not have to stack up a new predicate, given that there is none.
We now visit $b_{3-7}$, again, with an empty stack. Because the stack is empty, there are no links to create. But this block terminates with a branch $br b_0 \ell_8, \ell_{23}$. Thus, we push $b_0$ up. In this way, we will link $b_0$ to the instructions defined in the children of $b_{3-7}$. 

We now visit $b_{3-7}$, again, with an empty stack. Because the stack is empty, there are no links to create. But this block terminates with a branch $br b_0 \ell_8, \ell_{23}$. Thus, we push $b_0$ up. In this way, we will link $b_0$ to the instructions defined in the children of $b_{3-7}$. 

We now visit $b_{3-7}$, again, with an empty stack. Because the stack is empty, there are no links to create. But this block terminates with a branch $br b_0 \ell_8, \ell_{23}$. Thus, we push $b_0$ up. In this way, we will link $b_0$ to the instructions defined in the children of $b_{3-7}$.
We are now visiting the children of $b_{3-7}$. The order in which we visit these children is not important. Both are being visited with the predicate $\{b_0\}$. The important event that will take place soon is the linking of nodes...
We will create implicit flow edges (b₀, p₀) and (b₀, len₀). However, we will not create the edge (b₀, maxᵣ). This happens because b₂₃-₂₄ is the post-dominator of b₃-⁷, the block where b₀ is defined. When we visit b₂₃-₂₄, coming out of b₃-⁷, we pop b₀ before moving further down.
Once we visit $b_{10_{-}14}$, there are two things that we must do. First, we must link variables defined there to $b_0$, which is the current predicate. Second, we must update the current predicate to visit $b_{15_{-}16}$. Notice that this updating happens only to visit $b_{15_{-}16}$. We visit $b_{17_{-}18}$ with the old predicate $\{b_0\}$, because this block post-dominates $b_{10_{-}14}$.
We link every variable defined at block $b_{15-16}$ with $b_1$, because this is the current predicate once $b_{15-16}$ is visited. From $b_{15-16}$ we are done on this side of the tree, because this block has no children. Once we visit $b_{17-18}$ we link every instruction in this block to $b_0$, and not $b_1$, because $b_0$ is the current predicate when $b_{17-18}$ is visited. We do not have to stack $b_1$ up to visit $b_{17-18}$, because this block post-dominates $b_{10-14}$. 
After visiting $b_{19-19}$ and $b_{20-22}$ we are done. We link variables in $b_{19-19}$ to $b_2$, the current predicate, and we link variables in $b_{20-22}$ to $b_0$, the current predicate when that block is visited. Given that we have no more blocks to visit, we are done!
Contributions – I

- We avoid worst cases in the technique used by Ferrante et al. to handle nests of repeat loops and ladder CFGs.
Contributions – I

- We avoid worst cases in the technique used by Ferrante et al to handle nests of repeat loops (R) and ladder CFGs (L)
• We reduce the complexity of implementing Hunt and Sands' *System of Security Types*.

\[
\begin{align*}
\ell_1: & \quad w = \ast; \quad \ell_2: \quad x = \ast; \quad \ell_3: \quad y = \ast; \quad \ell_4: \quad z = \circ \\
\ell_5: & \quad p = \text{use}(x) \\
\ell_6: & \quad \text{branch } p \; \ell_9 \\
\ell_7: & \quad y = \text{use}(y); \quad \ell_8: \quad w = \text{use}(z) \\
\ell_9: & \quad p = \text{use}(x); \quad \ell_{10}: \quad \text{branch } p \; \ell_{\text{exit}} \\
\ell_{11}: & \quad z = \text{use}(z, w); \quad \ell_{12}: \quad x = \text{use}(z); \quad \ell_{13}: \quad z = \text{use}(x)
\end{align*}
\]
Contributions – II

• We reduce the complexity of implementing Hunt and Sands' *System of Security Types*.

\[
\begin{align*}
\ell_1 &: w_0 = \ast; \quad \ell_2 &: x_0 = \ast; \quad \ell_3 &: y_0 = \ast; \quad \ell_4 &: z_0 = \circ \\
\ell_5 &: p_0 = \text{use}(x_0) \\
\ell_6 &: \text{branch } p_0 \ell_9 \\
\ell_7 &: y_1 = \text{use}(y_0); \quad \ell_8 &: w_1 = \text{use}(z_0) \\
\ell_9 &: [y_2 = \phi(y_0, y_1); \quad w_2 = \phi(w_0, w_1)] \\
\ell_{10} &: [x_1 = \phi(x_0, x_2); \quad z_1 = \phi(z_0, z_3)] \\
\ell_{11} &: p_1 = \text{use}(x_1); \quad \ell_{12} &: \text{branch } p_1 \ell_{\text{exit}} \\
\ell_{13} &: z_2 = \text{use}(z_1, w_2); \quad \ell_{14} &: x_2 = \text{use}(z_2); \quad \ell_{15} &: z_3 = \text{use}(x_2)
\end{align*}
\]
Contributions – III

• We have detected security issues in real-world crypto libraries, such as TrueCrypt and OpenSSL.

Flow Tracker
Making programs safe through static flow analyses

• Define sources and sinks of information
• Finds paths between sources and sinks
• Available on-line at http://cuda.dcc.ufmg.br/flowtracker
Isochronous Code

- A program P is isochronous with regard to an input i if P always take the same time to execute, independent on the value of i.

```c
int foo(char* pw, char* in) {
    int i;
    for (i = 0; i < 3; i++) {
        if (pw[i] != in[i]) {
            return 0;
        }
    }
    return 1;
}
```

Is this program isochronous with regard to input pw?
Isochronous Programs

- A program P is isochronous with regard to an input i if P always take the same time to execute, independent on the value of i.

```cpp
int foo(char* pw, char* in) {
    int p0 = pw[0] == in[0];
    int p1 = pw[1] == in[1];
    int p2 = pw[2] == in[2];
    return p0 && p1 && p2;
}
```

What about this new program? Is it isochronous with regard to input p?
int foo(char* pw, char* in) {
    int p0 = pw[0] == in[0];
    int p1 = pw[1] == in[1];
    int p2 = pw[2] == in[2];
    return p0 && p1 && p2;
}
Compiling Isochronous Code

• Compiler writers usually do not worry about preserving the isochronous behavior of code.
• Yet, current tools that detect time-based side channels operate at the source code level.

We claim that this kind of information flow analysis should be performed at lower levels, for it is the implementation, not the abstract design of a crypto algorithm that will be running in production.
What FlowTracker does

- FlowTracker points out if secret information controls:
  - The outcome of conditional branches
  - The indices of memory addresses
Each tick in the X-axis is a different benchmark. FlowTracker inserts 1,11M control edges in gcc, our largest benchmark, which has 1.25M variables. Behavior is linear ($R^2 = 0.98$).
• **Address leak**: can an adversary infer the value of a pointer?
• **Buffer overflow**: can an adversary control the size of allocated memory, or indices of arrays?
• **Integer Overflow**: is there memory indexed by arithmetic operations that may overflow?
Issues in Real-World Code

• We have applied FlowTracker on GLS254‡.
  – Four warnings.
• Issues discovered by FlowTracker:
  – A timing-variant branch which depended on secret information flow but in practice was never taken.
    • The fix involved removing the branch altogether.
  – Variable-time code that was not called anywhere but was still pointed out.
    • The fix was deleting the affected code.
  – No issue found in NaCl

‡: Elliptic-curve Diffie-Hellman secret sharing using GLS binary curve \((L^2 + LZ + aZ^2)X^2 = X^4 + bZ^4\) defined over \(GF(2^{254})\) and implemented with lambda-projective coordinates \((X, L, Z)\).
Flow Tracker

Flow tracker is a tool that detects timing attack vulnerabilities in C/C++ programs that implement cryptographic routines. If the program contains a vulnerability, then Flow Tracker finds one or more (static) traces of instructions that uncover it. A vulnerable trace describes a sequence of data and control dependences from a secret information - e.g., a crypto key, the seed for a random number generator, etc - to a predicate of a branch or to a memory index. This is a problem because the branch normally influences the execution time of the program, and memory indexation may change the execution time of a program due to cache misses. So, if an adversary can measure this time for a given input which he controls, he or she may discover some information about the secret that we want to protect.
Type your code below:

```c
/*If pw == in returns 1, else returns 0 */
int compVar (char *pw, char *in) {
    int i;
    for (i=0; i<7; i++) {
        if (pw[i]!=in[i]) {
            return 0;
        }
    }
    return 1;
}

int main( void) {
    int comp=1;
    char pw["secret"];```

This page lets you run the Flow tracker and Ferrante analyses in order to analyze and compare their results.

Choose your tool:  

Choose the output:
- Standard output.  
- Pass statistics.  
- Table of contents.

You can also select a C/C++ file to upload.

[How to use this page.]
[Contact us.]
Conclusion

• Sparse implementation of information flows
• Track flow between values, not between instructions
• Web: http://cuda.dcc.ufmg.br/flowtracker
• Video tutorial: https://youtu.be/MunQ-C8yeIE

• Bruno Rodrigues - brunors172@gmail.com
• Diego Aranha - dfaranha@gmail.com
• Fernando Pereira - fernando@dcc.ufmg.br (that's me!)

We are looking for use-cases, and will be happy to help people to understand and use our tool