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

Bruno Rodrigues
UFMG, Brazil
brunors@dcc.ufmg.br

Fernando Magno Quintão Pereira
UFMG, Brazil
fernando@dcc.ufmg.br

Diego Aranha
UNICAMP, Brazil
dfaranha@ic.unicamp.br

Abstract

Information flow analyses traditionally use the Program Dependence Graph (PDG) as a supporting data-structure. This graph relies on Ferrante *et al.*'s notion of control dependences to represent implicit flows of information. A limitation of this approach is that it may create $O(|I| \times |E|)$ implicit flow edges in the PDG, where I are the instructions in a program, and E are the edges in its control flow graph. This paper shows that it is possible to compute information flow analyses using a different notion of implicit dependence, which yields a number of edges linear on the number of definitions plus uses of variables. Our algorithm computes these dependences in a single traversal of the program's dominance tree. This efficiency is possible due to a key property of programs in Static Single Assignment form: the definition of a variable dominates all its uses. Our algorithm correctly implements Hunt and Sands system of security types. Contrary to their original formulation, which required $O(I^2)$ space and time for structured programs, we require only $O(I)$. We have used our ideas to build FlowTracker, a tool that uncovers side-channel vulnerabilities in cryptographic algorithms. FlowTracker handles programs with over one-million assembly instructions in less than 200 seconds, and creates 24% less implicit flow edges than Ferrante *et al.*'s technique. FlowTracker has detected an issue in a constant-time implementation of Elliptic Curve Cryptography; it has found several time-variant constructions in OpenSSL, one issue in TrueCrypt and it has validated the isochronous behavior of the NaCl library.

Categories and Subject Descriptors D - Software [*D.3 Programming Languages*]: D.3.4 Processors - Compilers

General Terms Languages, Security, Experimentation

Keywords Information flow, implicit flows, sparse analyses, SSA

1. Introduction

Information flow analyses are used to prove properties such as confidentiality and integrity in sensitive software. Since the pioneering work of Denning and Denning [14], these analyses have been greatly expanded along theoretical and practical directions. Today, there exist purely static [14, 21], purely dynamic [48] and hybrid [20] ways to track the propagation of data along a program's code, and given the raising importance of security and privacy, such techniques are likely to evolve even more.

Implementations of flow analyses must track *explicit* and *implicit* propagation of information. There exists an explicit flow from a variable u to a variable v if v is defined by an instruction that uses u. Information runs implicitly from a predicate p to a variable v if p is used in a branch whose outcome controls an assignment into v, e.g.: "$v = p\ ?\ 0 : 1$". Estimating the explicit flows of information in a program is standard practice in the compiler's literature [14]. However, implicit flows are more elusive [40]. These flows are usually approximated via Ferrante *et al.*'s notion of *Control Dependences* [16, p.323] – a methodology that finds wide use in the literature [17–19, 36, 38, 41, 44, 46].

In Ferrante *et al.*'s work, explicit (e.g., data) and implicit (e.g., control) dependences are organized in a data-structured called the *Program Dependence Graph* (PDG). This abstraction has been originally designed to facilitate compilation tasks such as instruction scheduling and code parallelization [35]. Its use in the implementation of information flow analyses, although provably correct [17, 19, 36, 38, 46], has a shortcoming: it may create $O(|I| \times |E|)$ implicit flow edges in the PDG, where I are the instructions in a program, and E are the edges in its control flow graph. This high complexity is not a purely theoretical limit: it is possible to build actual worst-case scenarios, due to nests of do-while loops, or due to the so called ladder graphs [12, Fig.3]. Even though there are ways to compact control dependence edges [35], the quadratic-time complexity cannot be avoided if necessary to list all of them.

In this paper, we show that it is possible to compute information flow analyses using a different notion of implicit dependence, which avoids the worst-cases of Ferrante *et al.*'s approach. Whereas the idea of control dependence determines which instructions may, or may not *execute*, Denning&Denning-style [14] analyses are more concerned on which values can *influence* other values. In Section 3 we demonstrate that the Static Single Assignment (SSA) format [13], so popular in mainstream compilers, can lead to a natural definition of implicit dependences. In Section 3.1 we introduce an algorithm that determines all the implicit dependences in an SSA-form program in a single traversal of its dominance tree. We bound the number of implicit dependence edges in $O(|V|)$ for structured programs in SSA form, where V are the variables in the program. For non-structured SSA-form codes, our upper limit is $O(|V| + |U|)$, where $|U|$ is the number of uses of variables in the program. This complexity is still linear on the program size, measured as number of definitions plus uses of variables.

To validate our claims, we provide an implementation of Hunt and Sands flow-sensitive system of security types [21] in Section 4.2. This theoretical framework has been designed for While, a language common in information flow formalisms [21, 40]. We can handle non-structured codes, in addition to the structured syntax of While. Furthermore, by using SSA-form, we enable a *sparse*

analysis. An analysis is sparse if it associates information directly with variable names [11]. In contrast, a dense analysis associates information with pairs formed by variables and program points. The original work of Hunt and Sands fits this last category. Sparsity lets us reduce the complexity to store typing information from quadratic (on the number of variables or instructions) to linear.

This paper has practical contributions: in Section 5 we describe FlowTracker, an LLVM-based [25] tool that uses our implementation of Hunt&Sands type-system to uncover time-related side-channels in cryptographic algorithms. Timing attacks can be devastating against insecure implementations. For example, inexpensive timing attacks are very effective against naïve square-and-multiply implementations of RSA and Diffie-Hellman [24] or table-based software implementations of AES [32]. Presently, there is no automated technique that supports the validation of the time invariant behavior of compiled programs. This is a serious problem, as mainstream compilers are not guaranteed to preserve the isochronous behavior of algorithm whose source code has been validated against timing attacks. Our flow analysis is directly incorporated in the compiler and thus acts over its intermediate representation.

Section 5.1 reports experimental results obtained on widely used implementations of cryptography contained in NaCl [8], OpenSSL and GLS254 [31]. In particular, a known vulnerability in OpenSSL 1.01e was found by FlowTracker, suggesting that it could have been avoided before public discovery [47]. An issue found by FlowTracker in the record-setting GLS254 implementation of a binary elliptic curve key agreement protocol pointed developers to update their code. Our tool is industrial quality: it scales up remarkably well, as we show in Section 5.2. We have applied it onto all the SPEC CPU 2006 integer programs. We could analyze over 3.5 million assembly instructions in about 260 seconds.

2. Background

Information flow analyses try to infer properties of the data that a program manipulates. Typical implementations of flow systems associate program entities (variables, program points, branches, instructions) with points in a semi-lattice that represents levels of security. Without loss of generality, in this paper we assume that this semi-lattice has three elements: $\{H, L, \bot\}$, such that $H > L > \bot$. The (commutative) meet operator is defined as $H \wedge t = H, L \wedge \bot = L$, where $t \in \{H, L, \bot\}$. This theoretical framework is based on the work of Denning and Denning [14]. Their original information flow analysis is *flow-insensitive*. In other words, the result of analyzing a sequence of commands $C_1; C_2$ is the same as the result produced by the analysis of $C_2; C_1$. In terms of implementation, flow-insensitiveness implies that the type of a variable is determined by the most protective type that it receives anywhere in the program text.

A flow-insensitive information analysis is imprecise, because it does not consider the order in which variables are defined and used within a program. To address this shortcoming, in 2006 Hunt and Sands introduced a *flow-sensitive system* of security types [21]. In this system, the type of a variable depends on the program point where information is queried. Figure 1 illustrates this idea. Henceforth, we shall use • to denote a source of low-security information, and ∘ to denote a source of high-security information. Variable z is the only one initially bound to the type H in our program. In Hunt and Sands approach, we have a typing environment Γ binding variables to types at each *program point*. A program point, in this case, is any region between two instructions – a definition that includes the edges of the program's control flow graph.

If seen as an instance of the data-flow framework, Hunt and Sands' approach would be classified as a *dense* analysis [11, 30, 45]. A dense analysis associates pairs formed by variable names and program points with information. In Figure 1 we show the result of Hunt-Sands type inference for our example: this result uses $O(V^2)$ space. In contrast, a *sparse* analysis binds information directly to variable names [39]. A data-flow analysis must be applied on a program representation that presents the *Single Information Property*, to be implemented sparsely [45]. This property exists if the information that the analysis associates with a variable v is invariant at every program point where v is alive. Hunt and Sands have shown the existence of a program representation that lets them apply their type-system sparsely [21, Sec.7]. However, they did not provide a way to implement such an analysis; instead, they have shown that, given the result of their dense type-system, it is possible to derive a program representation that has the single information property. In Section 4.2, we show that their type-system can be implemented sparsely in programs on the Static Single Assignment format. Figure 2 (a) contains an SSA-form version of our running example, and the part (b) of the same figure shows a spoiler of our result: each variable has a unique type throughout its entire existence. Thus, our solution uses $O(V)$ space.

Tavares *et al.* [45] have provided a general approach to produce program representations that bestow the single information property to data-flow analyses. Their technique suits analyses in which all the information that contributes to the type of a variable is available at a certain program point. Even though we claim that the SSA form provides the single information property needed by Hunt-Sands technique, Tavares *et al*'s approach cannot be directly applied to implement that type-system. The problem in this case is due to *implicit flows*. We say that information runs implicitly from a predicate p to a variable v if p is used in a branch that *controls* the definition of v [40]. Here, we take Ferrante's definition of control dependence [16]:

Definition 2.1 (Ferrante'87) *A node y in a control flow graph is control dependent on a node x if: (i) there exists a path, in the control flow graph, from x to y, so that any node in this path is post-dominated by y. (ii) x is not post-dominated by y. In this case, we say that x* controls *the execution of y.*

As an example of control dependence, the branch at ℓ_6 controls the assignments at labels ℓ_7 and ℓ_8 in Figure 1. In Hunt-Sands approach, the type of variable w at ℓ_8 is determined as the meet of the types that z and p have at that point. This fact is unfortunate: because variable p is not syntactically present in ℓ_8, we cannot apply Tavares *et al*'s technique to generate a program rep-

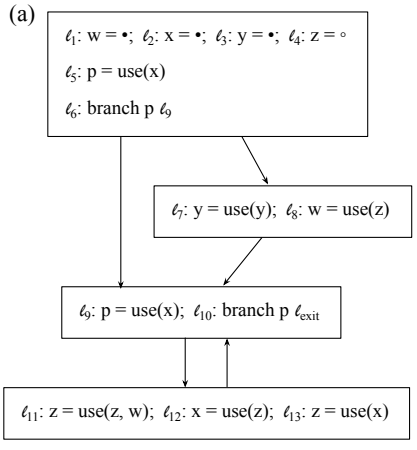

Figure 1. (a) Program taken from Hunt and Sands [21, Fig.3] rewritten using our low-level notation. (b) Type of each variable at each program point, as inferred using the flow-sensitive type-system of Hunt and Sands.

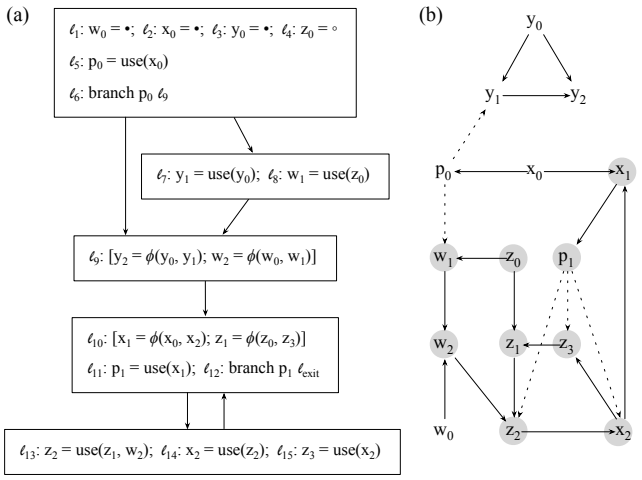

Figure 2. (a) Control flow graph from Figure 1 in the SSA format. (b) Flow graph that we build with the technique that we introduce in this paper. Nodes marked gray would have the H type in Hunt-Sands system [21].

resentation that "sparsifies" Hunt-Sands' analysis. Usually, when applied on low-level code, implementations of information flow analyses resort to Ferrante's algorithm [16], and its many improvements [12, 22, 23, 35] to find control dependences. We claim that this notion of control dependence, although correct (see, for instance, Hammer and Snelting [17]), gives more information than necessary to implement a flow-sensitive system of security types.

The program in Figure 3 illustrates this last point. This program contains a nest with three do-while loops. Figure 4 shows two different graphs that represent the possible ways in which information can flow between the variables in the code. These graphs contain a vertex for each program variable. A solid edge from u to v represents a *data-dependence*. Variable v is data-dependent on u if v is defined by an instruction that uses u. Data-dependencies create explicit flows of information. Dotted lines represent implicit flows. In Figure 4 (a), we use Ferrante's notion of control dependence to identify these flows, following previous work [17–19, 36, 38, 41, 44, 46]. Figure 4 (b) shows the implicit flows that we find using the algorithm that we introduce in this paper. To distinguish these lines from Ferrante's control dependence edges, we shall call them *implicit dependencies*.

Our concept of implicit dependence is fundamentally different than Ferrante *et al.*'s control dependences because we are interested in tracking which *values* – not which instructions – a predicate controls. As we will show in Section 3, a predicate controls the value of the ϕ-functions used in an SSA-form program. These special instructions let us join implicit flows. For instance, the label ℓ_7 in Figure 3 is control dependent – Ferrante style – from the label ℓ_9. However, the predicate p_1, which determines the outcome of the branch at ℓ_9, already determines the value of v_5, a variable defined by the ϕ-function at ℓ_6. Our algorithm will be able to explore this chain of transitive dependences: $p_1 \rightarrow v_7 \rightarrow v_5 \rightarrow v_6$; hence, avoiding the need to keep track of edges such as $p_1 \rightarrow v_6$.

The example of Figure 3 is an extreme case: nested sequences of do-while loops are well-known to produce a quadratic number of control dependence edges. The same is true for the so called "ladder graphs" [12]. The algorithm that we introduce in this paper does not suffer from these worst-case; nevertheless, for structured programs, our implicit dependences and Ferrante's control dependences tend to yield similar number of edges, although we can find them faster, as we demonstrate in Section 5.

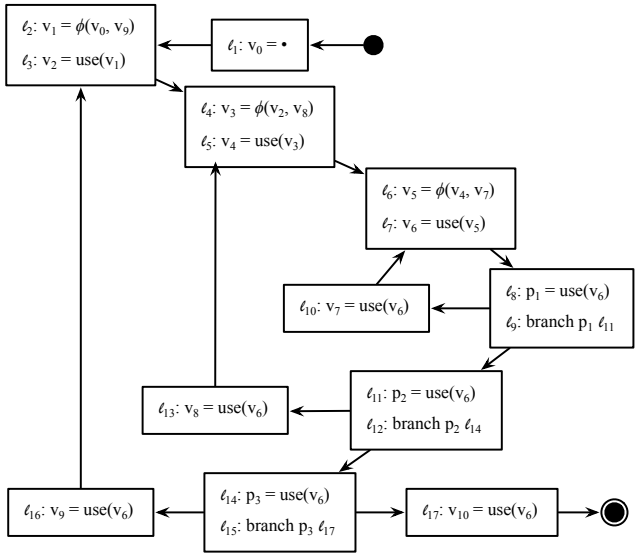

Figure 3. Control flow graph of three nested do-while blocks.

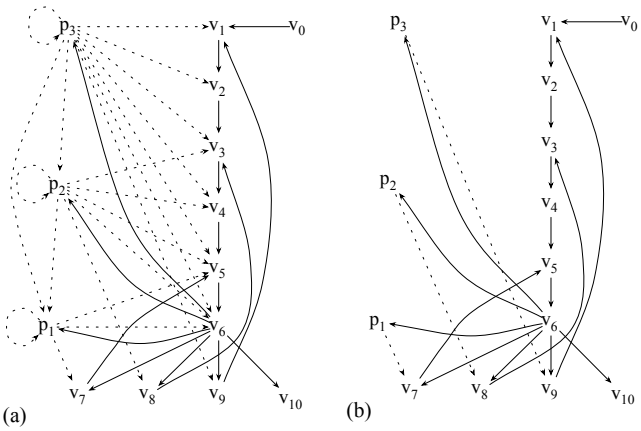

Figure 4. (a) Flow dependences as defined by Ferrante *et al.*. (b) Flow dependences as defined in this paper.

3. Information Flow in the SSA Graph

Basic Definitions. A *control flow graph* is a directed graph having two special nodes Start and End, so that every node is reachable from Start, no node reaches Start, every node reaches End, and End does not reach any node. Vertices in a control flow graph are called *Basic Blocks*. A block B' dominates another block B if every path from Start to B must go across B'. Dually, B *post-dominates* B' if every path from B' to End must pass through B. We let dom_B be the set of blocks that *dominate* block B, and pdom_B be the set of blocks that post-dominate B. If $B' \in \mathsf{dom}_B$, and for any other $B" \in \mathsf{dom}_B, B" \neq B'$ we have that $B' \in \mathsf{dom}_{B"}$, then we say that B' is the *immediate dominator* of B, written idom_B. We define the *immediate post-dominator* of B as the dual of the immediate dominator, and let it be pdom_B. The immediate dominator and the immediate post-dominator of any basic block in a control flow graph are unique [4]; thus, this two notions define trees: the *dominance tree* and the *post-dominance* tree.

A Core Language. We shall define our algorithms on top of the core language whose syntax is given in Figure 5. The semantics of this language will be defined in Section 4. A program is a sequence

Programs (*Prog*)	::=	$B_1, B_2, \ldots, B_n$
Basic Block (B)	::=	$P, I*, T$
Labels (L)	::=	$\{\ell_1, \ell_2, \ldots\}$
Variables (V)	::=	$\{v_1, v_2, \ldots\}$
Terminators	::=	
– Branch	|	$\ell : \texttt{br}(v, \ell_x)$
– Unconditional Jump	|	$\ell : \texttt{jmp}(\ell_x)$
ϕ-function (P)	|	$\ell : \overline{v} = \phi(\overline{v_1}, \ldots, \overline{v_n})$
Assignments (I)	::=	
– Low security data	|	$\ell : v = \bullet$
– High security data	|	$\ell : v = \circ$
– Sensitive operation	|	$\ell : v = \texttt{sink}(v)$
– Read operation	|	$\ell : v = \texttt{use}(v_1, \ldots, v_n)$

Figure 5. The syntax of our core language.

of basic blocks; basic blocks are sequences of instructions which end with either a branch or an unconditional jump. Each instruction is associated with a unique label. We recognize seven categories of instructions. Two of them (`br` and `jmp`) change the program's control flow. The others (ϕ-functions and the assignments) transfer information among variables. For simplicity, if the variable defined by `sink` is not used, we shall not show it in our examples.

3.1 Construction of Implicit Dependence Edges

We represent the flow dependences in a program via the *SSA Graph* [37]. The original definition of this data-structure is syntax-directed: it contains one vertex for each variable in the program text, and one edge from u to v if v appears on the left side of an instruction that contains u on its right side. The graphs in Figure 4 are SSA-graphs in the original sense, if we consider the solid edges only. The goal of this section is to augment these graphs with implicit dependence edges. Following the example of Figure 4, we shall produce the graph in the part (b). Figure 6 shows the algorithm that we use to add implicit dependence edges to the SSA graph. That algorithm, plus the notion of explicit dependences, lets us present our definition of the SSA graph:

Definition 3.1 (SSA Graph) *Given a program* Prog *written in the language seen in Figure 5, we define its SSA graph $G = (V, E)$ as follows: V contains one vertex for each variable v in the text of* Prog. *E contains an edge $v \rightarrow u$ if, and only if, u depends, explicitly or implicitly, from v. We say that a depends on b explicitly if b appears on the right side of the unique instruction that defines a. We say that a depends on b implicitly if the Algorithm of Figure 6 creates an edge from a to b.*

The algorithm in Figure 6 is written in Standard ML (SML). Empty brackets, e.g., [], denote an empty list. Double colons, ::, are list constructors. For instance, $(h :: t)$ is a list with head h, and tail t. The @ is list concatenation. Underline, _, represents any pattern. A construction such as "map (fn a => (lb, a) defList)" converts every element "a ∈ defList" into the pair "(lb, a)". We use a single opcode ASG to represent the four different kinds of assignments in our language. Figure 6 contains the entire algorithm, i.e., this program can be tested in an SML interpreter. We represent basic blocks as tuples. If a block B finishes with a conditional branch, then it has the format "BR (*Instructions*, *Predicate*, *Children*, $\texttt{pdom}_B$)". If B ends with an unconditional branch, then it has the format "JMP (*Instructions, Children*)". In either case, we let *Instructions* be the operations in the body of B and *Children* be the blocks that B dominates. If B is of the BR variety, then we let *Predicate* be the variable used in its branch. Figure 7 shows the data-structure that represents the program first seen in Figure 2.

The function `visit`, seen in Figure 6 goes down a program's dominance tree, keeping track of the last predicate seen during this traversal. By predicate we mean the variable `pred` used in a conditional branch. At a basic block B, `visit` invokes the function `link` to create an edge from `pred` to every variable defined in B. This invocation happens at lines 16, 18 and 26 of Figure 6. After inspecting a block B that ends in an instruction $\texttt{br}(\texttt{new\_pred}, \ell)$, `visit` is invoked recursively over every children of B in the dominance tree with `new_pred` as the current predicate, unless this children is the immediate post-dominator of B. If this last case happens, then `new_pred` is not used to replace `pred` as the current predicate. The application of `visit` on the structure seen in Figure 7 yields the five dashed edges that we have outlined in Figure 2 (b).

```
1  datatype Instruction =
2    ASG of string * string list
3    PHI of string list * string list list
4
5  datatype DomTree =
6    BR of Instruction list * string * DomTree list * DomTree
7  | JMP of Instruction list * DomTree list
8
9  fun link [] _ = []
10   | link _ "" = []
11   | link ((ASG (a, _)) :: insts) lb = (lb, a) :: link insts lb
12   | link ((PHI (defList, _)) :: insts) lb =
13                 (map (fn a => (lb, a)) defList) @ link insts lb
14
15 fun visit (JMP (instructions, [])) pred =
16           link instructions pred
17   | visit (JMP (instructions, [child])) pred =
18           link instructions pred @ visit child pred
19   | visit (BR (instructions, new_pred, children, ipdom)) pred =
20       let
21         fun visit_every [] = []
22           | visit_every (child::rest) =
23               if child = ipdom
24               then visit child pred @ visit_every rest
25               else visit child new_pred @ visit_every rest
26       in
27         link instructions pred @ visit_every children
28       end
```

Figure 6. The SML/NJ version of the basic algorithm that inserts implicit dependence edges in the SSA Graph.

```
1  val t_exit = JMP ([], [])
2  val t13_15 = JMP (
3     [ASG ("z2", ["z1", "w2"]), ASG ("x2", ["z2"]), ASG ("z3", ["x2"])], [])
4  val t10_12 = BR ([PHI (["x1", "z1"], [["x0", "x2"], ["z0", "z3"]]),
5       ASG ("p1", ["x1"])], "p1", [t13_15, t_exit], t_exit);
6  val t9_9 = JMP ([PHI (["y2", "w2"], [["y0", "y1"], ["w0", "w1"]])], [t10_12])
7  val t7_8 = JMP ([ASG ("y1", ["y0"]), ASG ("w1", ["z0"])], [])
8  val t1_6 = BR ([ASG ("w0", []), ASG ("x0", []), ASG ("y0", []),
9       ASG ("z0", []), ASG("p0", ["x0"])], "p0", [t7_8, t9_9], t9_9);
```

Figure 7. The data-structure that represents the program seen in Figure 2. To expression (`visit t1_6 ''`) invokes the algorithm of Figure 6 onto this data-structure.

3.2 Structural Properties of the Algorithm

We define the notion of *influence region* as follows:

Definition 3.2 (Influence Region) *The Influence Region IR_B of a block B that ends with a branch is a set of basic blocks. It contains block B' if, and only if: (i) $B \in \mathsf{dom}_{B'}$; (ii) $B' \notin \mathsf{pdom}_B$; (iii) there is no $B"$ such that $B" \in \mathsf{pdom}_B$ and $B" \in \mathsf{dom}_{B'}$.*

The *immediate influence region* of B is a subset of its influence region, which we denote by IIR_B, and define as:

Definition 3.3 (Immediate Influence Region) *The immediate influence region IIR_B of a block B contains blocks B' if, and only*

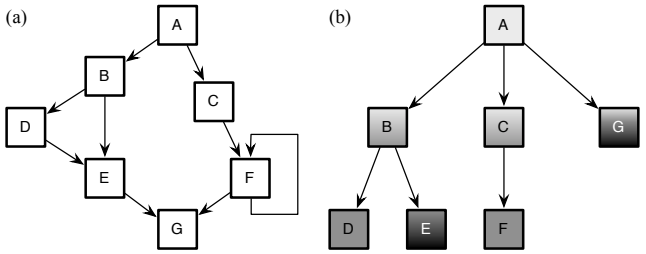

Figure 8. (a) Control flow graph. Each box represents a basic block. (b) Dominance tree.

if: (i) $B' \in \mathsf{IR}_B$; and (ii) there is no block $B"$, $B" \neq B$, such that $B' \in \mathsf{IIR}_{B"}$. We say that B is the head of IIR_B.

Figure 8 illustrates these definitions. $\mathsf{IR}_A = \{B, C, D, E, F\}$ and $\mathsf{IR}_B = \{D\}$. Block C does not have an influence region, because it does not terminate with a conditional branch. We have that $\mathsf{IIR}_A = \{B, C, F\}$ and $\mathsf{IIR}_B = \{D\}$. The notions of influence region and immediate influence region have geometrical interpretations. The influence region of a block B is every node that is reachable from B on a traversal of the program's dominance tree, stopping at any node in pdom_B. The immediate influence region of B is given by a traversal of the dominance tree, again starting at B, and stopping at any node that post-dominates B, and also at any node B' that ends in a branch (inclusive, e.g., $B' \in \mathsf{IIR}_B$ as long as $B' \notin \mathsf{pdom}_B$). The algorithm from Figure 6 links a predicate p to the variables defined within the immediate influence region of the block where p is used.

Our algorithm from Figure 6 creates a chain of transitive dependences that links a predicate with all the variables whose assignment this predicate controls. However, this statement is only true if we guarantee that every predicate is defined immediately before being used in a branch. To ensure this property it suffices to replace every conditional test like "$\mathtt{br}(v, \ell)$" by two instructions in sequence: "$v' = v; \mathtt{br}(v', \ell)$", where v' is a fresh variable which will be used only at the branch. This transformation happens in constant time per branch; hence, it is linear on the number of conditionals that exist in a program. Notice that the programs that we have seen in Figures 1, 2 and 3 are already in this format, i.e., every predicate is defined immediately before the branch where it is used. Theorem 3.4 states the key invariant property of our algorithm.

Theorem 3.4 *If a basic block B ends with a conditional "$\mathtt{br}(p, \ell)$", then function $\mathtt{visit}$ creates a chain of implicit dependences linking p to every variable defined within IR_B.*

3.3 Dealing with Non-Structured Codes and Codes in Non-Conventional SSA Form

Due to reasons that we discuss in Section 4.1, our algorithm must create a chain of dependence edges between a predicate p, used in a branch such as $\mathtt{br}(p, \ell)$, and every variable defined by a ϕ-function whose outcome p controls. This property will not hold if the arguments of a ϕ-function are not defined in the influence region of p. Such situation may happen, for instance, in *non-structured* programs, and in programs in *non-conventional static single assigment* (CSSA) form.

Following Ferrante *et al.* [16], a *Hammock Region* within a control flow graph is the set of blocks between a branch and its post-dominator. We shall say that a program is non-structured if it contains at least one hammock region R that can be entered from different blocks outside R. For instance, the program in Figure 9 (a) is non-structured. A program P is in Conventional Static Single Assignment (CSSA) form if it does not contain two *ϕ-related*

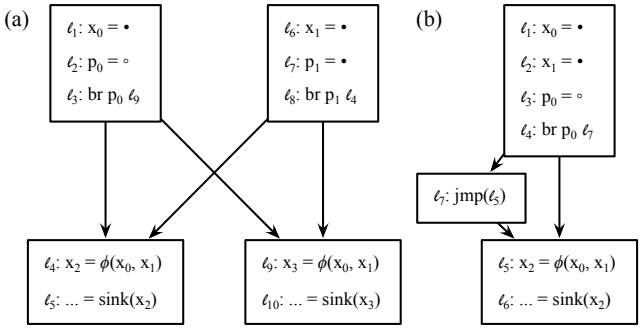

Figure 9. (a) Non-structured control flow graph. (b) Program in non-conventional SSA form.

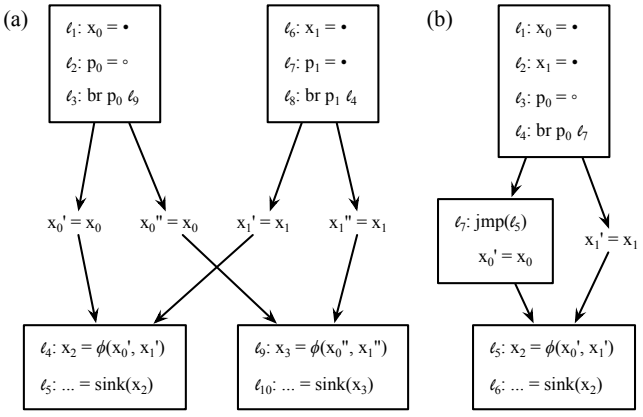

Figure 10. Programs from Figure 9, after live range splitting à la Sreedhar [43].

variables simultaneously alive at the same program point. Quoting Pereira and Palsberg [34], "two variables, v_1 and v_2, are ϕ-related in an SSA form program if either (i) they are used in the same ϕ-function, as parameters or definition, or (ii) there exists a third variable v_3, so that v_1 and v_2 are ϕ-related to v_3." The program in Figure 9 (b) is not in CSSA form, because ϕ-related variables x_0 and x_1 overlap.

The algorithm in Figure 6 will not create any implicit dependence edge in either program of Figure 9. This fact is unfortunate, for in both cases the predicates p_0 and p_1 control de outcome of the ϕ-functions. There are two ways to deal with this fact: we can change our algorithm to add control edges between a predicate p and the ϕ-functions that exist at the *Dominance Frontier* (for the meaning of Dominance Frontier, see Definition 3.5) of the block where p is used; or we can *split live ranges*, following the approach proposed by Sreedhar *et al.*'s [43] to convert an SSA-form program to the CSSA representation. We use this last strategy, because it simplifies our correctness proofs.

The *live range* of a variable v is the set of program points where v is *alive*. In an SSA-form program, v is alive at every point that is reachable from a backwards traversal of this program's CFG, starting at a site where v is used, and stopping at the – unique – site where v is defined [4, ch.15]. To implement our live range splitting strategy, we: (i) break critical edges[1], and (ii) split live ranges of the arguments of ϕ-functions at the predecessors of the node where these ϕ-functions are located. We split the live range of a variable v

[1] A CFG edge is critical if it links a basic block with multiple successors to a basic block with multiple predecessors.

at a CFG edge $\ell_i \to \ell_j$ by (i) inserting a copy $v' = v$ at the edge, v' being a fresh variable name; (ii) renaming every use of v dominated by ℓ_j to v' and (iii) inserting ϕ-functions in the program to re-create the SSA property. Because we use Sreedhar *et al.*'s method [43], step (iii) is not necessary. Figure 10 illustrates this process. We split the live ranges of those variables that we call *escaping*; we define escaping variables as follows:

Definition 3.5 (Escaping Variable) *We say that v escapes the influence region of a branch "ℓ_b : $\texttt{branch}(p, \ell)$", located at label ℓ_b if, and only if v is alive at an edge $\ell_o \to \ell_d$, such that: (i) ℓ_o is dominated by ℓ_b; and (ii) ℓ_d is not dominated by ℓ_b. In compiler's jargon, we say that edge $\ell_o \to \ell_d$ is in ℓ_b's* dominance frontier.

The live range splitting strategy that we have just presented enables our core algorithm to establish dependences between the predicate p that controls a branch, and every ϕ-function whose outcome may be controlled by such a branch. We state this property formally in Theorem 3.6.

Theorem 3.6 *After live range splitting, we ensure that the core algorithm creates a path of dependence edges between a predicate p used at "$\texttt{branch}(p, \ell')$", and every variable defined by a ϕ-function whose outcome this branch controls.*

Complexity. Function `visit` creates at most one implicit dependence edge leading to any vertex in the SSA-graph of a program. This result lets us put a complexity bound on the space requirements of the algorithm that we introduce in this paper. Our complexity results require variables in the SSA format. SSA-form programs contain more variables than their original versions; however, this growth is still linear [42]. Sreedhar's live range splitting strategy, which we use to deal with non-structured codes, creates a new variable for each argument of a ϕ-function. Hence, `visit` might create one implicit dependence edge per variable in the SSA-form program, and one such edge per argument of a ϕ-function. This is still linear on the *size* of the original program, measured as the number of definitions and uses of variables. Furthermore, Benoit *et al.* have shown that a variable will rarely be used more than five times in real-world benchmarks [10]. Thus, our algorithm is likely to be linear on the number of program variables in practice, even for non-structured codes. Notice that we are using the naïve method introduced by Sreedhar *et al.* (Method I) [43, Sec4.1]. In the same paper, they have discussed two other ways to split live ranges (Method II and III), which introduce fewer new variable names.

4. Semantic Properties

In this section, we discuss the guarantees that our algorithm provides to its users. To this end, Figures 11, 12 and 13 define the semantics of the language seen in Figure 5. The relation $\xrightarrow{i}$ (Figure 12) describes the semantics of data and arithmetic operations, and the relation $\xrightarrow{e}$ (Figure 13) gives us the semantics of operations that change the program's control flow. State is given by a triple:

- `pc`: the program counter, which indexes a vector I of instructions. This element is only used in the relation $\xrightarrow{e}$.
- S: the stack of local variables, which binds names to values. The possible values are either $\bullet$ or $\circ$. We define $\sqcup$ so that $\bullet \sqcup \bullet = \bullet$ and $\circ \sqcup x = x \sqcup \circ = \circ$ for any x.
- P: the set of *active* predicates. We say that a predicate p is active if the program flow is traversing the influence region of a basic block that ends with $\texttt{br}(p, \ell)$.

Again, we let the double colon (::) denote list construction. Assignments add a new binding on top of the stack S. Whenever necessary to retrieve the value x associated with a variable v, we scan the stack, from top towards bottom, looking for a pair (v, x). This search is performed by the function **lookup**, seen in Figure 11.

$$\textbf{lookup}([(v,x) :: S], v) \Rightarrow x \qquad \frac{v' \neq v \qquad \textbf{lookup}(S, v) \Rightarrow x}{\textbf{lookup}([(v', x') :: L], v) \Rightarrow x}$$

$$\frac{v \in S_v}{\textbf{lookupFirst}([(v,x) :: S], S_v) \Rightarrow x}$$

$$\frac{v' \notin S_v \qquad \textbf{lookupFirst}(S, S_v) \Rightarrow x}{\textbf{lookupFirst}([(v', x') :: S], S_v) \Rightarrow x}$$

$$\frac{\forall v_i \in P, \textbf{lookup}(S, v_i) = x_i \qquad x_1 \sqcup \ldots \sqcup x_n = x}{\textbf{join}(S, P) \Rightarrow x}$$

Figure 11. Stack management library.

$$P \vdash \langle v = \circ, S\rangle \xrightarrow{i} (v, \circ) : S$$

$$P \vdash \langle v = \bullet, S\rangle \xrightarrow{i} (v, \textbf{join}(S, P)) : S$$

$$\frac{\textbf{lookup}(S, v') \Rightarrow \bullet \qquad \textbf{join}(S, P) = \bullet}{P \vdash \langle v = \texttt{sink}(v'), S\rangle \xrightarrow{i} (v, \bullet) : S}$$

$$\frac{\textbf{lookupFirst}(S, \{v_1, v_2\}) \Rightarrow x}{P \vdash \langle v = \phi(v_1, v_2), S\rangle \xrightarrow{i} (v, x) : S}$$

$$\frac{\textbf{lookup}(S, v_i) \Rightarrow x_i \qquad x = x_1 \sqcup \ldots \sqcup x_n \sqcup \textbf{join}(S, P)}{P \vdash \langle v = \texttt{use}(v_1, \ldots, v_n), S\rangle \xrightarrow{i} (v, x) : S}$$

Figure 12. Semantics of data and arithmetic operations.

$$\frac{I[\texttt{pc}] = \texttt{jmp}(\ell) \qquad I \vdash \langle \ell, S, P\rangle \xrightarrow{e} \langle S', P'\rangle}{I \vdash \langle \texttt{pc}, S, P\rangle \xrightarrow{e} \langle S', P'\rangle}$$

$$\frac{I[\texttt{pc}] = \texttt{br}(v, \ell) \qquad I \vdash \langle \ell, S, P \cup \{v\}\rangle \xrightarrow{e} \langle S', P'\rangle}{I \vdash \langle \texttt{pc}, S, P\rangle \xrightarrow{e} \langle S', P'\rangle}$$

$$\frac{I[\texttt{pc}] = \texttt{br}(v, \ell) \qquad I \vdash \langle \texttt{pc} + 1, S, P \cup \{v\}\rangle \xrightarrow{e} \langle S', P'\rangle}{I \vdash \langle \texttt{pc}, S, P\rangle \xrightarrow{e} \langle S', P'\rangle}$$

$$\frac{I[\texttt{pc}] = \texttt{pdom}(P") \qquad I \vdash \langle \texttt{pc} + 1, S, P \setminus P"\rangle \xrightarrow{e} \langle S', P'\rangle}{I \vdash \langle \texttt{pc}, S, P\rangle \xrightarrow{e} \langle S', P'\rangle}$$

$$\frac{P \vdash \langle I[\texttt{pc}], S\rangle \xrightarrow{i} S' \qquad I \vdash \langle \texttt{pc} + 1, S', P\rangle \xrightarrow{e} \langle S", P"\rangle}{I \vdash \langle \texttt{pc}, S, P\rangle \xrightarrow{e} \langle S", P"\rangle}$$

Figure 13. The operational semantics of instructions that change the program's flow of control. Conditional branches are non-deterministic: any outcome is valid to our purposes.

When interpreting a ϕ-function such as $v = \phi(v_1, v_2)$, we search the stack for the first occurrence of any binding containing either v_1 or v_2. This action is implemented by the function **lookupFirst**, also seen in Figure 11. Search for the first parameter of a ϕ-function is correct for strict programs, i.e., programs in which every variable is defined before being used. This is one of the core properties of SSA-form programs, as stated by Zhao *et al* [49]. For simplicity, we shall assume that every basic block contains at most one ϕ-function at its beginning. To handle multiple ϕ-instructions, Zhao *et al.* search for a list of arguments, instead of just one.

To model the effects of $\ell : \texttt{br}(v, \ell')$ over assignments that happen inside the influence region of ℓ we use the environment of active predicates P. Whenever we traverse that branch, we add v to P. The effects of v cease at $\texttt{pdom}_\ell$, for then we are back into a program region that will necessarily be traversed if we visit ℓ, independent on the value of v. Contrary to the While programming languages used in standard security type systems [21, 40], our toy language does not offer syntax to indicate post-dominance. Thus, we shall assume a special instruction $\ell_p : \texttt{pdom}(P')$ at label $\ell_p = \texttt{pdom}_\ell$. We let P' be the set of variables used in branches that are post-dominated by ℓ_p. As we see in Figure 13, this instruction removes from the set of active predicates every variable in P.

4.1 From Dependences to Types

If a program I, in a state $\langle S, P\rangle$ cannot take a step, then we say that this program is *stuck*. There are several ways in which programs in our language can be stuck. Our abstract machine is stuck if: (i) we have a program counter pointing to a non-existing instruction (see Figure 13); (ii) we have the use of a non-defined variable (see Figure 12); or (iii) `sink` receives an argument bound to the value $\circ$. Because we are not interested in malformed control flow graphs, we shall rule out (i). Zhao *et al.* [49] have shown that (ii) cannot happen in strict SSA-form programs, i.e., programs in which any use of a variable is dominated by its definition. Programs that do not present conditions (i) or (ii) are said to be *well-formed*. Henceforth, we will only consider well-formed programs. Thus, we shall assume that only condition (iii) can stop our abstract machine from progressing. Notice that we do not model termination in our semantics: the existence of an instruction `halt` is immaterial to our formalization. The explicit dependences, plus the implicit dependences created by the procedure seen in Figure 6 lets us define a way to assign types to variables:

Definition 4.1 (Types as Reachability in the SSA Graph) *Let* Prog *be a program, $G = (V, E)$ its SSA graph, and $V_\circ$ be the set formed by the variables v defined as $v = \circ$ in* Prog. *If $v \in V_\circ$, then v has type H. The type of a variable u reachable from any variable $v \in V_\circ$ through a traversal of G is H; otherwise, it is L.*

Notice that Definition 4.1 is dealing with a security lattice of height two. We could use more expressive lattices by adapting Definition 4.1 accordingly, e.g., the type of a variable v is the meet over all paths of the types of all the variables that reach v. We shall restrict the discussions in the rest of this section to the simpler lattice. Theorem 4.2 states our notion of non-interference.

Theorem 4.2 *A variable of type L cannot be assigned a value $\circ$.*

Progress follows from Theorem 4.2:

Corollary 4.3 *[Progress] If a well-formed program does not contain an instruction $v' = \texttt{sink}(v)$ such that v has type H, then this program cannot be stuck.*

Corollary 4.3, e.g., Progress, means that a sink instruction will not receive a value of type H. On the other hand, it says nothing about the execution of sink instructions within program regions of high security level. As pointed out by Russo and Sabelfeld [40], events like this can also leak sensitive information. Fortunately, it is easy to adapt our semantics to model this kind of indirect leaking. Every sink operation defines a variable, e.g., $\ell : v = \texttt{sink}(v')$. The security level at ℓ is given by the type of v.

4.2 Equivalence with Hunt-Sands Type-System

Our notions of explicit and implicit dependences gives us an efficient way to implement Hunt and Sands [21] type-system. To support this statement, Figure 14 provides a translation of the

$CS[\![\textbf{skip}]\!] \xrightarrow{t} ;$	$CS[\![\mathrm{v_s} :^{t}= \mathrm{E}^{\Gamma}]\!] \rightarrow \mathrm{I};\ \mathrm{v_s} = \texttt{use}(x)$
$CS[\![\mathrm{D_1}; \mathrm{D_2}]\!] \xrightarrow{t}$ $CS[\![\mathrm{D_1}]\!]; CS[\![\mathrm{D_2}]\!]$	$CS[\![$ **if** E^{Γ} $(\mathrm{D_1}; \boldsymbol{\Gamma}' = \boldsymbol{\Gamma}_1) \xrightarrow{t} \ell : CS[\![\mathrm{D_1}]\!];$ $(\mathrm{D_2}; \boldsymbol{\Gamma}' = \boldsymbol{\Gamma}_2) \quad CS[\![\mathrm{D_2}]\!];$ $]\!]$ I; $\texttt{br}(x, \ell)$; $\boldsymbol{\Gamma}' = \phi(\boldsymbol{\Gamma}_1, \boldsymbol{\Gamma}_2)$

$CS[\![\boldsymbol{\Gamma}' = \boldsymbol{\Gamma}_0;$ **while** E^{Γ} $(\mathrm{D_n}; \boldsymbol{\Gamma}' = \boldsymbol{\Gamma}_n)\ ]\!] \xrightarrow{t}$
ℓ_0: $\boldsymbol{\Gamma}' = \phi(\boldsymbol{\Gamma}_1, \boldsymbol{\Gamma}_2)$; I; $\texttt{br}(x, \ell_n)$; $CS[\![\mathrm{D_n}]\!]$; $\texttt{jmp}(\ell_0)$; ℓ_n:

Figure 14. Translation of the high-level While language with fixed types [21, Fig.4] to our low-level language. We translate every expression E^{Γ} as a sequence of instructions I that produces a final result into a variable x.

While language into our core language seen in Figure 5. Our starting point is the version of While with fixed types that Hunt and Sands have defined in Section 7 of their work. As Hunt and Sands have speculated [21, Sec7.6], this version is very close to the Static Single Assignment form. Thus, we perform the translation by joining, via ϕ-functions, the variables that they have defined at the end of control flow constructs. Hunt and Sands have not defined the semantics of While; however, Russo and Sabelfeld have done it posteriorly [40]. Hence, in order to show the correctness of the translation, we use this latter formalization. In Theorem 4.4, we let the relation $\xrightarrow{r}$ represent the steps defined by Russo and Sabelfeld for their version of the While language.

Theorem 4.4 *Let C be a statement of While such that $CS[\![C]\!] \xrightarrow{t} I$, and let S be a stack. We have that if $\langle C, S\rangle \xrightarrow{r} \langle stop, S'\rangle$ then there exists an active set P' such that $I \vdash \langle \ell_0, S, \{\}\rangle \xrightarrow{e} \langle S', P'\rangle$.*

We can show that our algorithm, when applied on the low-level language that we obtain via the compiler seen in Figure 14, correctly implements Hunt & Sands typing rules. We formalize this statement in Theorem 4.5. Without loss of generality, we assume a type-system with only two types, e.g., H and L. To generalize this type-system to higher semi-lattices, we let the type of a variable v be the meet-over-all-paths of the types of every variable that can influence v. Again, variable u influences variable v if, and only if, there exists a path in the SSA graph from u to v.

Theorem 4.5 *Let C be a statement of While such that $CS[\![C]\!] \xrightarrow{t} I$. The Hunt & Sands type-system assigns variable $v_x \in C$ a type H, if and only if, v_x has type H according to definition 4.1.*

5. FlowTracker

To validate all the ideas discussed in this paper, we have materialized them into FlowTracker, a flow-sensitive, interprocedural static analyzer. This software can be adapted to track different types of information: the user is free to define sources and sink operations. Interprocedurality is implemented through summary edges, following a description by Hammer and Snelting [17]. Since March of 2014, we have been using FlowTracker to detect time-based side-channels in crypto implementations. Following Agat [2], we recognize two types of time-based information leaks. In the first category, we group disclosures that happen when secret data determine which parts of the program are executed. The second category encompasses code in which memory is indexed by sensitive information. Figure 15 provides an example of each kind of vulnerability. Our example consists of a simple function, which receives a password encoded as an array of characters `pw`, and tries to match this string against another array `in`, which represents the input provided

by an external user. In this example, we regard the user input as malicious, for it could have been tainted by an adversary.

Leaks due to control flow. The program in Figure 15 (a) contains a time-based information disclosure vulnerability. In this example, an attacker can time how long it takes for function `isDiffVul1` to return. An earlier return indicates that the match at line 4 failed in one of the first characters. By varying, in lexicographic order, the contents of array `in`, the adversary can reduce the complexity of the secret search problem from exponential to linear.

Leaks due to cache behavior. The program in Figure 15 (b) is an attempt to remove the time-based side-channel from the program seen in Figure 15 (a). Function `isDiffVul2` uses a table to check if the characters used in the sensitive password, .e.g., array `pw` match those present in the input array `in`. If all the characters in both strings appear in the same order, the function returns true, otherwise it returns false. The secret `pw` does not control any branch in function `isDiffVul2`; however, this code still presents a timing-based leak. Data belonging to the password is used to index memory at line 6 of our example. Depending on the relative distance between the characters of `pw`, cache misses may happen. In this case, an adversary can obtain information about how spaced out are the alphanumeric elements present in `pw`. The feasibility of this kind of attack has been demonstrated in previous work [6].

Removing the time-based side-channels. Figure 16 shows a sanitized version of our string matching example. In this case, independent on the secret value stored in `pw`, function `isDiffOk` has the same control flow. In other words, each path within the program will be traversed the same number of times, and in the same order, regardless of the input. Furthermore, function `isDiffOk` will index the same blocks of memory, always in the same order, and always with the same strides between successive accesses, independent on its inputs. In this case, we say that secret information does not influence neither control flow, nor memory indexation. FlowTracker detects the two vulnerabilities seen in Figures 15 (a) and (b), and reports no false positive for the program shown in Figure 16.

Compiler Induced Leaks. As observed in Section 1, a compiler may introduce time-based side channels when operating on source code thought to be isochronous. A typical example is due to the short-circuiting operators from C and Java. One could be tempted to believe that the code in Figure 17 (a) is time-invariant. However, the compiler will translate this code into the version seen in the part (b) of the same Figure. Thus, we believe that techniques that detect time-variant behaviour at the source code level [2], although very useful during the process of software development, must be complemented by validation at the compiler level. For instance, quoting Agat [2, p.51]: "A minor oversimplification in our semantics is that it neglects the unconditional jumps made in the machine code that a compiler produces for if- and while- commands". Our lower-level approach does not suffer from such limitation. Notice that FlowTracker is more restrictive than the work of Agat, because it verifies that "there is no branching or memory indexation on high-security data". We chose this more restrictive approach because Aciicmez *et al.* [1] has shown that allowing balanced branches as Agat does still gives room for branch prediction attacks.

```
1 int isDiffVul1
      (char *pw, char *in) {
2   int i;
3   for (i=0; i<7; i++) {
4     if (pw[i]!=in[i]) {
5       return 0;
6     }
7   }
8   return 1;
9 }
```

(a)

```
1 int isDiffVul2
      (char *pw, char *in) {
2   int i;
3   int isDiff = 0;
4   char array[128] = { 0 };
5   for (i=0; i<7; i++) {
6     array[pw[i]] += i + 1;
7   }
8   for (i=0; i<7; i++) {
9     array[in[i]] -= i + 1;
10  }
11  for (i=0; i<128; i++) {
12    isDiff |= array[i];
13  }
14  return isDiff;
15 }
```

(b)

Figure 15. (a) Program whose control flow is controlled by secret information. (b) Program that may leak timing information due to cache behavior.

```
1  #define F(i) diff |= pw[i] ^ in[i]
2
3  int isDiffOk(char *pw, char *in) {
4    int diff = 0;
5-12 F(0); F(1); F(2); F(3); F(4); F(5); F(6); F(7);
13   return (1 & ((diff - 1) >> 8)) - 1;
14 }
```

Figure 16. Isochronous implementation of functions `isDiffVul1` and `isDiffVul2` in Figure 15.

```
int eq(char *p, char *q) {
  a0 = (p[0] == q[0]);
  a1 = (p[1] == q[1]);
  a2 = (p[2] == q[2]);
  return a0 && a1 && a2;
}
```

(a)

```
int eq(char *p, char *q) {
  if (p[0] != q[0])
    return false;
  else if (p[1] != q[1])
    return false;
  else
    return p[2] == q[2];
}
```

(b)

Figure 17. (a) Isochronous program. (b) Time variant code.

5.1 Effectiveness

We have evaluated the effectiveness of FlowTracker in two different ways: by outsourcing it through an on-line server (`http://cuda.dcc.ufmg.br/flowtracker/`) and by applying it on well-known cryptographic libraries such as NaCl v20110221, TrueCrypt or parts of OpenSSL. Concerning the first approach, we have made FlowTracker available since March of 2014. From the iteration with external users we got several benchmarks, ranging from trivial programs, such as the one in Figure 16, to complex code, such as the X25519 curve-based key agreement used in LibSSH [7] or the benchmarks of Eldib *et al.* [15]. FlowTracker correctly reported time-based leaks in all the examples where they were expected to be found, and did not trigger any warning in the cases where warnings were not expected. Some of these results, such as those obtained on the QMS benchmarks [15] have not been reported before.

We have applied FlowTracker on popular implementations of cryptography, starting from NaCl, because of its constant-time features [8]. NaCl contains implementations of several cryptographic primitives, including hash functions, message authentication codes (MACs), authenticated encryption, digital signatures and public key encryption. Besides the obvious secret and private keys, hash function inputs and encryption plaintext messages were conservatively marked as sensitive. As expected, the isochronous properties of the entire library were formally verified and no vulnerabilities were found, confirming previous results [3]. The analysis spanned 12 reference implementations in the C programming language, encompassing 45 different functions and over 6,000 lines of code: SHA2-based HMAC, Salsa20 stream cipher variants, Poly1305 authenticator, Curve25519 [7] and their combinations.

We have applied FlowTracker on several OpenSSL functions. Contrary to NaCl, in this case we got several warnings. Due to the multiple interfaces for the same primitives, the analysis was restricted to the security-critical operations required by RSA and Elliptic Curve Cryptography, namely modular exponentiation and scalar multiplication. FlowTracker has reported 1,217 warnings in 6 OpenSSL binary/prime elliptic curve scalar multiplication and

Montgomery exponentiation functions. An example of vulnerable implementation was scalar multiplication by Montgomery laddering in a binary curve (function `ec_GF2m_montgomery_point_multiply()` in file `ec2_mult.c`) from OpenSSL version 1.0.1e. The side-channel susceptibility of this code was recently demonstrated by a Flush+Reload cache-timing attack [47], corroborating our findings. Running FlowTracker in the newer 1.0.2 version verified that the changes applied by the OpenSSL team solved the specific vulnerability, but 82 vulnerable subgraphs still remain in the function, despite the natural side-channel resistance of the underlying algorithm [26]. Thus, we claim that integrating FlowTracker in the development process would have detected the vulnerability and prevented release in production-ready code.

We have not forced time-based attacks on all the warnings issued by FlowTracker – this kind of verification is too time consuming. However, even a warning that cannot be exploited in practice already indicates poor coding patterns. For instance, we have applied FlowTracker onto the GLS254 implementation [31]. FlowTracker pointed 4 issues in the code, but manual inspection did not suggest any clear attack vector. More precisely, a critical loop inside the scalar recoding procedure depends on the precision of part of the private key. This precision is fixed with overwhelming probability due to a mathematical result, but this fact cannot be determined automatically by our tool. After reporting FlowTracker's warning, the code was fixed and no information leaks are reported in the most current version of the software. Moreover, even if no direct vulnerability was found, the constant-time property in the updated version is now much easier to verify by manual inspection.

5.2 Scalability

To probe the efficiency and the scalability of FlowTracker, we have applied it onto the SPEC CPU 2006 programs, and in the C programs available in the LLVM test suite. These programs are not used in crypto applications. However, they are large enough to give us an idea on: (i) how much time FlowTracker needs to build the dependence graph of large programs; (ii) how much memory FlowTracker requires, and (iii) what is the relation between the size of programs and the size of their SSA graphs. Similar experiments would not be as meaningful if run on the crypto algorithms of Section 5.1, because those programs are much smaller. The numbers that we report in this section have been obtained on an Intel I7 with a 2.20GHz clock, eight cores, and 8GB of RAM. Each core has a 6,144KB cache. Our machine uses Linux Ubuntu 12.04.

Figure 18 shows the size of the SSA graphs produced for SPEC. We have roughly 1.5x more data dependence edges than program variables and about 2.5 variables per control dependence edge. Our algorithm is 7% faster than Ferrante *et al.*'s. These numbers are the average of three separate runs, and include the time to find implicit and explicit dependence edges. The latter is the same for both algorithms. Runtime variance is negligible, and we omit it. The time to build the dominance and post-dominance trees is a small fraction of the time necessary to build the graphs. This happens because the algorithms that build those trees work at the granularity of basic blocks, whereas the algorithms that build the SSA graphs work at the granularity of instruction operands. In other words, they visit each operand of each instruction in the program.

Figure 18 indicates that our graphs are sparse, i.e., the number of edges that they have is linearly proportional to the number of nodes. We have performed experiments to demonstrate this last fact. To this end, we have applied FlowTracker on the 50 largest benchmarks in the LLVM test suite. The smallest program (`GlobalDataFlow-flt`) has 9,124 LLVM instructions, and the largest (`consumer-typeset`) has 261,492. Figure 19 summarizes this experiment. The coefficient of determination between time and size of dependence graph is 0.97, which indicates strong linear correlation. In total, these fifty programs contain 2,044,883 instructions, in their LLVM intermediate representation. Our SSA graphs contain a total of 4,718,821 data dependence edges, and 1,730,219 control dependence edges. The total time to build all these graphs was 133.51 seconds. This number includes the time to parse the programs, build their dominance and post-dominance trees, plus the time to generate the SSA graph.

Bench	Vars	Dep	Time	FDep	FT	D/PD
perl	434	239	19.3	338	22.0	0.8/0.7
bzip2	27	21	1.1	25	1.1	0.1/0.1
mcf	4	1	0.1	2	0.1	0.0/0.0
gobmk	234	102	3.3	137	3.3	0.1/0.1
hmmer	108	53	1.1	71	1.1	0.2/0.1
sjeng	43	22	0.5	34	0.5	0.1/0.1
libq	9	3	0.1	4	0.1	0.0/0.0
h264r	234	111	5.3	140	5.3	0.3/0.3
omnet	161	32	1.2	39	1.2	0.3/0.3
astar	13	5	0.1	7	0.1	0.0/0.0
xalanc	1,038	261	48.9	301	46.6	2.1/1.9
gcc	1,249	912	181.8	1,208	203.6	2.0/1.9
Total	3,560	1,767	262.6	2,311	281.5	6.8/6.2

Figure 18. How FlowTracker scales up in the SPEC benchmark suite compared to the classic algorithm of Ferrante *et al.* [16]. **Vars** ($\times 1,000$): number of variables in each program (equals the number of vertices in the SSA graph); **Dep** ($\times 1,000$): number of control dependence edges that FlowTracker creates; **Time** (sec): FlowTracker's runtime; **FDep** ($\times 1,000$): number of control dependence edges created by Ferrante *et al.*; **FT** (sec): Ferrante *et al.*'s runtime. **D/PD** (sec): time to build dominance and post-dominance trees. Ferrante's algorithm uses only post-dominance; we use both.

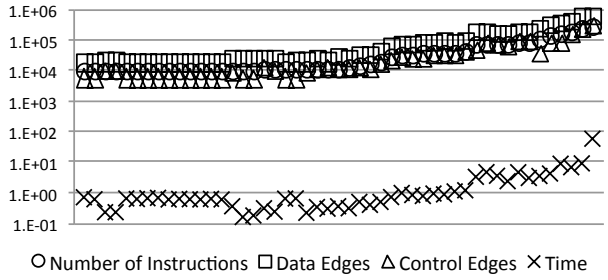

Figure 19. Size of programs (Number of Instructions) vs size of SSA graph (number of control and data dependence edges) vs time (sec) to build the SSA graph. Each tick in the X-axis is a different benchmark. Benchmarks are sorted by number of instructions.

Both algorithms, Ferrante *et al.*'s and ours, produce the same number of data dependence edges; however, we generate fewer edges to track implicit dependences. Figure 18 shows that we have produced 24% less implicit dependence edges than Ferrante *et al.* for the programs in SPEC CPU 2006. Furthermore, our algorithm does not suffer from some pathological cases that are possible in Ferrante *et al.*'s. These worst-case scenarios are caused, for instance, by nests of repeat loops, as seen in Figure 3. Ladder control flow graphs also produce many control dependences in programs [12]. Figure 20 demonstrates this fact. We have generated programs with varying number of syntactic features, e.g., steps in ladder CFGs, or nests of repeat loops, and applied both algorithms onto these programs. Whereas the number of edges that we create grows linearly, Ferrante *et al.*'s algorithm shows quadratic behavior.

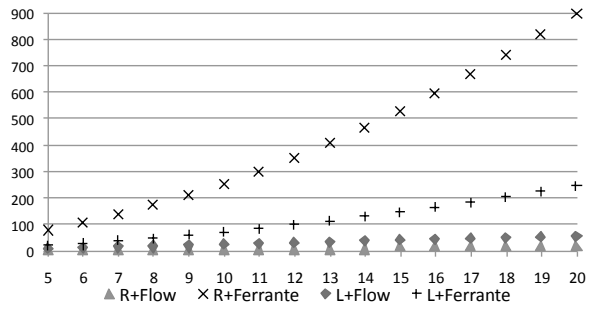

Figure 20. Comparison between our algorithm and Ferrante *et al.*'s for nests of repeat loops and ladder CFGs. **R**: repeat loops; **L**: ladder graphs; **Flow**: FlowTracker's algorithm; **Ferrante**: the algorithm seen in [16]. Y-axis indicates number of implicit dependence edges. X-axis indicates either number of nested repeat loops, or number of steps in the ladder CFG.

6. Related Works

Ferrante *et al.*'s notion of dependence graph [16], has been improved along different directions. These improvements speedup the construction of the graph, or make it more space-efficient. To save construction time, Johnson and Pingali [23] have introduced the *Program Structure Tree* (PST), which represents hammock regions in programs, and, consequently, gives us a cheap way to determine control dependences between instructions. In terms of space, Cytron and Ferrante [12] have introduced the *Control Dependence Graph*, which tracks control dependences more compactly. In a similar direction, Pingali and Bilardi [35] have designed the *Augmented Post-Dominator Tree*, a structure that allows the efficient spatial representation of implicit dependences. Nevertheless, in spite of all these advances, Ferrante *et al.*'s notion of control dependences still can produce a quadratic number of edges denoting implicit flows of information. Thus, if necessary to list all of them, as in the construction of dependence graphs, it is possible to build worst-cases where the quadratic behavior surfaces.

None of these approaches rely on SSA properties to achieve more simplicity or efficiency. That is one of the greatest contributions of this paper. Although simplicity is a subjective concept, we claim that we have a simpler approach than previous work based on two observations. First, our **executable** prototype of Figure 6 fits into 27 lines of SML code. Even though its LLVM counterpart is larger – about 600 lines of C++ – this extra size is mostly necessary to handle the different kinds of instructions in the LLVM IR. Second, much of the previous literature on dependence tracking claims as a contribution the fact that they do not require the dominance tree of a program. Yet, this data-structure is available for free in mainstream compilers, such as gcc, LLVM, Jikes and icc, because it is used in the construction of the SSA format. By avoiding them, the previous algorithms renounce useful information.

Ottenstein *et al.*'s [33] GSA form is the work that is the closest to ours. The GSA form links ϕ-functions to the predicates that control them. We do not link predicates and ϕ-functions; however, we might link predicates with the parameters of ϕ-functions whenever these parameters are defined in the influence region of said predicates. Nevertheless, we are solving a different problem with a very different algorithm. Ottenstein *et al.* want to build an "interpretable" flavour of SSA form. We want to track control dependences. GSA is not the format that we use, because it has not been conceived as a way to implement a system of security types.

Techniques to detect and prevent timing leaks. We have used the techniques discussed in this paper to uncover time-based side channels. Related methodologies and guidelines to prevent such vulnerabilities abound. John Agat has proposed a type-system to transform a vulnerable program into a secure one. He performs this transformation by inserting dummy instructions in branch blocks, to mitigate the runtime difference between the two paths that can be taken out of a conditional test. Molnar *et al.* [29] have designed and implemented a source-to-source C translator that detects and fixes control-flow based leakages. Lux *et al.* [28] have implemented a tool that detects timing attack vulnerabilities in Java programs. A key difference between our work and all this previous literature is the fact that they operate on a high-level programming language, using a set of inference idioms similar to the rules that Hunt and Sands [21] have proposed for the While language. We claim that our approach has a few advantages, because it works directly on the compiler's intermediate representation. Namely, we can deal with different programming languages and do not need to rely on the assumption that the compiler is not introducing time-based leaks into the executable code. There exist information flow frameworks that have been used in low-level languages, mostly Java bytecodes [5, 9, 17, 27]. These tools have not been customized to detect time-based side channels. Thus, to the best of our knowledge, Section 5.1 contains the first report concerning the search of time-related information leaks at the compiler level. Nevertheless, we believe that those tools previously cited could be adapted to serve such purpose without any fundamental change in their core algorithms. Unfortunately, the descriptions of these algorithms do not state in details which technique they use to track implicit flows.

7. Conclusion

This paper has presented a novel way to track the propagation of information on low-level, potentially non-structured, codes. The key idea behind the efficiency of our algorithm is the use of Static Single Assignment form. Contrary to previous works that rely on Ferrante *et al.*'s notion of control dependence, we are interested in knowing which values can be affected by the flow of information, not which instructions may or may not execute depending on how information propagates on the program. This notion of influence is conceptually closer to previous work on information flow, such as Hunt and Sands' [21] or Agat's [2]. Our idea of implicit dependences comes with a number of benefits. First, the SSA-based approach leads naturally to sparse – instead of dense – implementations of information flow analyses. For instance, we reduce the space and time complexity taken from Hunt and Sands' type-system from quadratic to linear on the number of program variables. Second, our algorithm avoids some quadratic worst-case scenarios that are possible in the method introduced by Ferrante *et al.* to discover control dependences. To demonstrate the effectiveness of our algorithm, we have used it to implement FlowTracker, a tool that discovers time-based side-channels in cryptographic code. FlowTracker has been used successfully to analyze several publicly available libraries of cryptography, such as OpenSSL and NaCl.

Software FlowTracker has an on-line interface: `http://cuda.dcc.ufmg.br/flowtracker/`. The interested reader can visualize the dependence graphs produced by our algorithm and Ferrante's at `http://cuda.dcc.ufmg.br/flowtrackervsferrante`

Acknowledgment

This project is supported by the Brazilian Ministry of Science and Technology through CNPq, by FAPEMIG, by FAPESP and by the Intel Corporation through the Intel University Research Office. We thank David Ott and his colleagues from Intel for reading an early draft of this paper. We are indebted to the CC referees for sending us useful comments and suggestions to improve this work.

References

[1] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. In *ASIACCS*, pages 312–320. ACM, 2007.

[2] J. Agat. Transforming out timing leaks. In *POPL*, pages 40–53. ACM, 2000.

[3] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 78(7):796–812, 2013.

[4] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[5] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. *Mathematical Structures in Computer Science*, 23(5):1032–1081, 2013.

[6] D. J. Bernstein. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.

[7] D. J. Bernstein. Curve25519: new diffie-hellman speed records. In *PKC*, pages 207–228. Springer, 2006.

[8] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *Progress in Cryptology – LATINCRYPT*, pages 159–176. Springer, 2012.

[9] G. Bian, K. Nakayama, Y. Kobayashi, and M. Maekawa. Java bytecode dependence analysis for secure information flow. *I. J. Network Security*, 4(1):59–68, 2007.

[10] B. Boissinot, S. Hack, D. Grund, B. D. de Dinechin, and F. Rastello. Fast liveness checking for SSA-form programs. In *CGO*, pages 35–44. IEEE, 2008.

[11] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66. ACM, 1991.

[12] R. Cytron, J. Ferrante, and V. Sarkar. Compact representations for control dependence. In *PLDI*, pages 337–351. ACM, 1990.

[13] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[14] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20:504–513, 1977.

[15] H. Eldib, C. Wang, M. Taha, and P. Schaumont. QMS: Evaluating the side-channel resistance of masked software from source code. In *DAC*, pages 209:1–209:6. ACM, 2014.

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.

[17] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Secur.*, 8(6):399–422, 2009.

[18] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *ISOLA*, pages 119–128. IEEE, 2006.

[19] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *POPL*, pages 146–157. ACM, 1988.

[20] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, pages 40–51, 2004.

[21] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL*, pages 79–90. ACM, 2006.

[22] R. Johnson and K. Pingali. Dependence-based program analysis. In *PLDI*, pages 78–89. ACM, 1993.

[23] R. Johnson, D. Pearson, and K. Pingali. The program tree structure. In *PLDI*, pages 171–185. ACM, 1994.

[24] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113. Springer, 1996.

[25] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[26] J. López and R. Dahab. Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation. In *CHES*, volume 1717 of *LNCS*, pages 316–327. Springer, 1999.

[27] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber. Cassandra: Towards a certifying app store for android. In *SPSM*, pages 93–104. ACM, 2014.

[28] A. Lux and A. Starostin. A tool for static detection of timing channels in java. *Journal of Cryptographic Engineering*, 1(4):303–313, 2011.

[29] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC*, pages 156–168. Springer, 2006.

[30] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for c-like languages. In *PLDI*, pages 1–11. ACM, 2012.

[31] T. Oliveira, J. López, D. F. Aranha, and F. Rodríguez-Henríquez. Two is the fastest prime: lambda coordinates for binary elliptic curves. *J. Cryptographic Engineering*, 4(1):3–17, 2014.

[32] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA*, pages 1–20. Springer, 2006.

[33] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*, pages 257–271. ACM, 1990.

[34] F. M. Q. Pereira and J. Palsberg. SSA elimination after register allocation. In *CC*, pages 158 – 173, 2009.

[35] K. Pingali and G. Bilardi. Optimal control dependence computation and the roman chariots problem. In *TOPLAS*, pages 462–491. ACM, 1997.

[36] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *TOPLAS*, 29(5), 2007.

[37] F. Rastello. *SSA-based Compiler Design*. Springer, 1st edition, 2015.

[38] T. Reps and W. Yang. The semantics of program slicing. Technical report, University of Wisconsin – Madison, 1988.

[39] A. Rimsa, M. d'Amorim, F. M. Q. Pereira, and R. da Silva Bigonha. Efficient static checker for tainted variable attacks. *Sci. Comput. Program.*, 80:91–105, 2014.

[40] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, pages 186–199. IEEE Computer Society, 2010.

[41] G. Snelting. Combining slicing and constraint solving for validation of measurement software. In *SAS*, pages 332–348. Springer, 1996.

[42] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing ϕ-nodes. In *POPL*, pages 62–73. ACM, 1995.

[43] V. C. Sreedhar, R. D. ching Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *SAS*, pages 194–210. Springer, 1999.

[44] M. Taghdiri, G. Snelting, and C. Sinz. Information flow analysis via path condition refinement. In *FAST*, pages 65–79. Springer, 2011.

[45] A. Tavares, B. Boissinot, F. Pereira, and F. Rastello. Parameterized construction of program representations for sparse dataflow analyses. In *CC*, pages 2–21. Springer, 2014.

[46] D. Wasserrab, D. Lohner, and G. Snelting. On pdg-based noninterference and its modular proof. In *PLAS*, pages 31–44. ACM, 2009.

[47] Y. Yarom and N. Benger. Recovering openssl ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. `http://eprint.iacr.org/`.

[48] R. Zhang, S. Huang, Z. Qi, and H. Guan. Combining static and dynamic analysis to discover software vulnerabilities. In *IMIS*, pages 175–181. IEEE Computer Society, 2011.

[49] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of ssa-based optimizations for llvm. In *PLDI*, pages 175–186. ACM, 2013.