# Securing a Compiler Transformation

Chaoqiang Deng[1] and Kedar S. Namjoshi[2(✉)]

[1] New York University, New York City, USA
deng@cs.nyu.edu
[2] Bell Laboratories, Nokia, Murray Hill, USA
kedar@research.bell-labs.com

**Abstract.** A compiler can be correct and yet be insecure. That is, a compiled program may have the same input-output behavior as the original, and yet leak more information. An example is the commonly applied optimization which removes dead (i.e., useless) stores. It is shown that deciding *a posteriori* whether a new leak has been introduced as a result of eliminating dead stores is difficult: it is PSPACE-hard for finite-state programs and undecidable in general. In contrast, deciding the correctness of dead store removal is in polynomial time. In response to the hardness result, a sound but approximate polynomial-time algorithm for secure dead store elimination is presented and proved correct. Furthermore, it is shown that for several other compiler transformations, security follows from correctness.

## 1 Introduction

Compilers are essential to computing: without some form of compilation, it is not possible to turn a high level program description into executable code. Ensuring that a compiler produces correct code – i.e., the resulting executable has the same input-output behavior as the input program – is therefore important and a classic verification challenge. In this work, we assume correctness and investigate the *security* of a compiler transformation.

Compilers can be correct but insecure. The best-known example of this phenomenon is given by dead store elimination [7,10]. Consider the program on the left hand side of Fig. 1. It reads a password, uses it, then clears the memory containing password data, so that the password does not remain in the clear on the stack any longer than is necessary. Dead-store elimination, applied to this program, will remove the instruction clearing `x`, as its value is never used. In the resulting program, the password remains in the clear in the stack memory, as compilers usually implement a return from a procedure simply by moving the stack pointer to a different position, without erasing the procedure-local data. As a consequence, an attack elsewhere in the program which gains access to program memory may be able to read the password from the stack memory. Stated differently, the value of the password is leaked outside the function `foo`. The input-output behavior of the function is identical for the two programs, hence the dead-store removal is correct.


X. Rival (Ed.): SAS 2016, LNCS 9837, pp. 170–188, 2016.
DOI: 10.1007/978-3-662-53413-7_9

```
void foo()
{
  int x;

  x = read_password();
  use(x);
  x = 0; // clear password
  return;
}
```

```
void foo()
{
  int x;

  x = read_password();
  use(x);

  return;
}
```

**Fig. 1.** C programs illustrating the insecurity of dead-store elimination

There are workarounds which can be applied to this example to fix the problem. For example, `x` could be declared volatile in `C`, so the compiler will not remove any assignments to `x`. Specific compiler pragmas could be applied to force the compiler to retain the assignment to `x`. But these are all unsatisfactory, in several respects. First, they pre-suppose that the possibility of a compiler introducing a security leak is known to the programmer, which may not be the case. Next, they suppose that the programmer understands enough of the compiler's internal workings to implement the correct fix, which need not be the case either – compilation is a complex, opaque process. Furthermore, it supposes that the solution is portable across compilers, which need not be true. And, finally, the fix may be too severe: for instance, an assignment `x:= 5` following the clearing of `x` can be removed safely, as the `x:= 0` assignment already clears the sensitive data – marking `x` as volatile would prohibit this removal. A similar issue arises if a compiler inserts instructions to clear all potentially tainted data at the return point; as taint analysis is approximate, such instructions may incur a significant overhead. For these reasons, we believe it is necessary to find a fundamental solution to this problem.

One possible solution is to develop an analysis which, given an instance of a correct transformation, checks whether it is secure. This is a *Translation Validation* mechanism for security, similar to those developed in e.g., [12,15,18] for correctness. We show that translation validation for information leakage is undecidable for general programs and difficult (PSPACE-hard) for finite-state programs. This proof considers a dead store elimination transformation where the input and output programs as well as the location of the eliminated stores is supplied. On the other hand, given the same information, correctness can be determined in polynomial time. The large complexity gap suggests that translation validation for information leakage is likely to be much more difficult than the corresponding question for correctness.

Faced with this difficulty, we turn to algorithms that guarantee a *secure* dead-store elimination. Our algorithm takes as input a program $P$ and a list of dead assignments, and prunes that list to those assignments whose removal is guaranteed not to introduce a new information leak. This is done by consulting the result of a control-flow sensitive taint analysis on the source program $P$. We formalize the precise notion of information leakage, present this algorithm,

and the proof that it preserves security. Three important points should be noted. First, the algorithm is sub-optimal, given the hardness results. It may retain more stores than is strictly necessary. Second, the algorithm does not eliminate leaks that are originally in $P$; it only ensures that no new leaks are added during the transformation from $P$ to $Q$. Thus, it shows that the transformation is secure in that it does not weaken the security guarantees of the original program. Finally, we assume correctness and focus on information leakage, which is but one aspect of security. There are other aspects, such as ensuring that a compiler does not introduce an undefined operation, e.g., a buffer overrun, which might compromise security. That is part of the correctness guarantee.

The difference between correctness and security is due to the fact that standard notions of refinement used for correctness do not necessarily preserve security properties. We develop a strong notion of refinement that does preserve security, and use it to show that other common optimizations, such as constant propagation, are secure.

To summarize, the main contributions of this work are (1) results showing that *a posteriori* verification of the security of compilation has high complexity, (2) a dead-store elimination procedure with security built in, along with a formal proof of the security guarantees that are provided, and (3) a general theorem which reduces security to correctness through a strong refinement notion. These are first steps towards the construction of a fully secure compiler.

## 2 Preliminaries

In this section, we define the correctness and the security of transformations on a small programming language. The programming language is deliberately kept simple, to more clearly illustrate the issues and the proof arguments.

*Program Syntax and Semantics.* For the formal development, we consider only **structured** programs whose syntax is given in the following. (Illustrative examples are, however, written in C.) For simplicity, all variables have Integer type. Variables are partitioned into input and state variables and, on a different axis, into sets $H$ ("high security") and $L$ ("low security"). All state variables are low security; inputs may be high or low security.

$x \in \mathbb{X}$ variables

$e \in \mathbb{E} ::= c \mid x \mid f(e_1, \ldots, e_n)$ expressions: $f$ is a function, c a constant

$g \in \mathbb{G}$ Boolean conditions on $\mathbb{X}$

$S \in \mathbb{S} ::= \mathsf{skip} \mid x := e \mid S_1; S_2 \mid \mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid \mathsf{while}\ g\ \mathsf{do}\ S\ \mathsf{od}$ statements

A program can be represented by its *control flow graph* (CFG). (We omit a description of this process, which is standard.) Each node of the CFG represents a program location, and each edge is labeled with a guarded command, of the form "$g \rightarrow x := e$" or "$g \rightarrow \mathsf{skip}$", where $g$ is a Boolean predicate and $e$ is an expression over the program variables. A special node, entry, with no incoming

edges, defines the initial program location, while a special node, exit, defines the final program location. Values for input variables are specified at the beginning of the program and remain constant throughout execution.

The semantics of a program is defined in the standard manner. A *program state* $s$ is a pair $(m, p)$, where $m$ is a CFG node (referred to as the *location* of $s$) and $p$ is a function mapping each variable to a value from its type. The function $p$ can be extended to evaluate an expression in the standard way (omitted). We suppose that a program has a fixed initial valuation for the state variables. An *initial state* is one located at the entry node, where the state variables have this fixed valuation. The *transition relation* is defined as follows: a pair of states, $(s = (m, p), t = (n, q))$ is in the relation if there is an edge $f = (m, n)$ of the CFG which connects the locations associated with $s$ and $t$, and for the guarded command on that edge, either (i) the command is of the form $g \rightarrow x := e$, the guard $g$ holds of $p$, and the function $q(y)$ is identical to $p(y)$ for all variables $y$ other than $x$, while $q(x)$ equals $p(e)$; (ii) the command is of the form $g \rightarrow \mathsf{skip}$, the guard $g$ holds of $p$, and $q$ is identical to $p$. The guard predicates for all of the outgoing edges of a node form a partition of the state space, so that a program is *deterministic* and *deadlock-free*. A *execution trace* of the program (referred to in short as a trace) from state $s$ is a sequence of states $s_0 = s, s_1, \ldots$ such that adjacent states are connected by the transition relation. A *computation* is a trace from the initial state. A computation is *terminating* if it is finite and the last state has the exit node as its location.

*Post-domination in CFG.* A set of nodes $N$ *post-dominates* a node $m$ if each path in the CFG from $m$ to the exit node has to pass through at least one node from $N$.

*Information Leakage.* Information leakage is defined in a standard manner [3,6]. Input variables are divided into high security ($H$) and low security ($L$) variables. All state variables are low-security ($L$). A program $P$ is said to *leak* information if there is a pair of $H$-input values $\{a, b\}$, with $a \neq b$, and an $L$-input $c$ such that the computations of $P$ on inputs $(H = a, L = c)$ and $(H = b, L = c)$ either (a) differ in the sequence of output values, or (b) both terminate, and differ in the value of one of the $L$-variables at their *final* states. We call $(a, b, c)$ a *leaky triple* for program $P$.

*Correct and Secure Transformations.* For clarity, we consider program transformations which do not alter the set of input variables. A transformation from program $P$ to program $Q$ may alter the code of $P$ or the set of state variables. The transformation is *correct* if, for every input value $a$, the sequence of output values for executions of $P$ and $Q$ from $a$ is identical. (The return value from a function is considered to be an output.) The transformation is *secure* if the set of leaky triples for $Q$ is a subset of the leaky triples for $P$. By the definition of leakage, for a correct transformation to be insecure, there must be a triple $(a, b, c)$ for which the computations of $Q$ with inputs $(H = a, L = c)$ and $(H = b, L = c)$ terminate with different $L$-values, and either one of the corresponding computations in $P$ is non-terminating, or both terminate with the same $L$-values.

A correct transformation supplies the relative correctness guarantee that $Q$ is at least as correct as $P$, it does not assure the correctness of either program with respect to a specification. Similarly, a secure transformation does not ensure the absolute security of either $P$ or $Q$; it does ensure relative security, i.e., that $Q$ is not more leaky than $P$.

This definition of a secure transformation does not distinguish between the "amount" of information that is leaked in the two programs. Consider, for instance, the case where both $P$ and $Q$ leak information about a credit card number. In program $Q$, the entire card number is made visible whereas, in $P$, only the last four digits are exposed. By the definition above, this transformation is secure, as both programs leak information about the credit card number. From a practical standpoint, though, one might consider $Q$ to have a far more serious leak than $P$, as the last four digits are commonly revealed on credit card statements. It has proved difficult, however, to precisely define the notion of "amount of information" – cf. [16] for a survey. We conjecture, however, that the dead-store elimination procedure presented in this paper would not incur greater amount of information leakage than the original program; a justification for this conjecture is presented in Sect. 5.

## 3 The Hardness of Secure Translation Validation

One method of ensuring the security of a compiler transformation would be to check algorithmically, during compilation, that the result program $Q$ obtained by the transformation on program $P$ is at least as secure as $P$. This is akin to the Translation Validation approach [12,15,18] to compiler correctness. We show, however, that checking the security of a program transformation is hard and can be substantially more difficult than checking its correctness. Focusing on the dead store elimination procedure, we show that checking its security is undecidable in general, PSPACE-complete for finite-state programs, and co-NP-complete for loop-free, finite-state programs.

The precise setting is as follows. The input to the checker is a triple $(P, Q, D)$, where $P$ is an input program, $Q$ is the output program produced after dead store elimination, and $D$ is the list of eliminated assignments, which are known to be dead (i.e., with no useful effect) at their locations. The question is to determine whether $Q$ is at most as leaky as $P$. To begin with, we establish that checking correctness is easy, in polynomial time, for arbitrary **while** programs.

**Theorem 1.** *The correctness of a dead store elimination instance* $(P, Q, D)$ *can be checked in* PTIME.

**Proof:** The check proceeds as follows. First, check that $P$ and $Q$ have the same (identical, not isomorphic) graph. I.e., the set of node names is identical, and the transition relation is identical. Then check if $Q$ differs from $P$ only in the labeling of transitions in $D$, which are replaced by skip. Finally, check that every store in $D$ is (syntactically) dead in $P$, by re-doing the liveness analysis on $P$. Each step is in polynomial time in the size of the programs. **EndProof.**

We now turn to the question of security, and show that it is substantially more difficult.

**Theorem 2.** *Checking the security of a dead store elimination given as a triple* $(P, Q, D)$ *is* PSPACE*-complete for finite-state programs.*

**Proof:** Consider the complement problem of checking whether a transformation from $P$ to $Q$ is insecure. By definition, this is so if there exists a triple $(a, b, c)$ which is leaky for $Q$ but not for $P$. Determining whether $(a, b, c)$ is leaky can be done in deterministic polynomial space, by simulating the program on the input pairs $(a, c)$ and $(b, c)$ independently in parallel. Checking the pairs sequentially does not work, as the computation from one of the pairs may not terminate. Non-termination is handled in a standard way by adding an $n$-bit counter, where $2^n$ is an upper bound on the size of the search space: the number $n$ is linear in the number of program variables. A non-deterministic machine can guess the values $a, b, c$ in polynomial time, and then check that $(a, b, c)$ is leaky for $Q$ but not leaky for $P$. Thus, checking insecurity is in non-deterministic PSPACE, which is in PSPACE by Savitch's theorem.

To show hardness, consider the problem of deciding whether a program with no inputs or outputs terminates, which is PSPACE-complete by a simple reduction from the IN-PLACE-ACCEPTANCE problem [13]. Given such a program $R$, let $h$ be a fresh high security input variable and $l$ a fresh low-security state variable, both Boolean, with $l$ initialized to *false*. Define program $P$ as: "$R$; $l := h$; $l :=$ *false*", and program $Q$ as: "$R$; $l := h$". As the final assignment to $l$ in $P$ is dead, $Q$ is a correct result of dead store elimination on $P$. Consider the triple $(h = \mathit{true}, h = \mathit{false}, \_)$. If $R$ terminates, then $Q$ has distinct final values for $l$ for the two executions arising from inputs $(h = \mathit{true}, \_)$ and $(h = \mathit{false}, \_)$, while $P$ does not, so the transformation is insecure. If $R$ does not terminate, there are no terminating executions for $Q$, so $Q$ has no leaky triples and the transformation is trivially secure. Hence, $R$ is non-terminating if, and only if, the transformation from $P$ to $Q$ is secure. **EndProof.**

**Theorem 3.** *Checking the security of a dead store elimination given as a triple* $(P, Q, D)$ *is undecidable for general programs.*

**Proof:** (Sketch) The PSPACE-hardness proof of Theorem 2 can be applied to general programs as well. Hence, the triple is insecure if, and only if, program $R$ terminates. **EndProof.**

In the full version of the paper, we show that establishing security is difficult even for the very simple case of finite-state, loop-free programs.

**Theorem 4.** *Checking the security of a dead store elimination given as a triple* $(P, Q, D)$ *is co-*NP*-complete for loop free programs.*

## 4 A Taint Proof System

In this section, we introduce a taint proof system for structured programs. It is similar to the security proof systems of [6,17] but explicitly considers per-variable, per-location taints. It is inspired by the taint proof system of [4], which is the basis of the STAC taint analysis plugin of the Frama-C compiler. There are some differences in the treatment of conditionals: in their system, assignments in a branch of an IF-statement with a tainted condition are tainted in an eager fashion while, in ours, the taint may be delayed to a point immediately after the statement.

The full version of this paper includes a proof of soundness for this system. Moreover, the key properties carry over to a more complex taint proof system for arbitrary CFGs. Although the focus here is on structured programs, this is done solely for clarity; the results carry over to arbitrary CFGs.

### 4.1 Preliminaries

Let $Taint$ be a Boolean set $\{untainted, tainted\}$ where $tainted$ = true and $untainted$ = false. A taint environment is a function $\mathcal{E} : Variables \to Taint$ which maps each program variable to a Boolean value. That is, for a taint environment $\mathcal{E}$, $\mathcal{E}(x)$ is true if $x$ is tainted, and false if $x$ is untainted. The taint environment $\mathcal{E}$ can be extended to terms as follows:

$\mathcal{E}(c)$ = false, if $c$ is a constant
$\mathcal{E}(x) = \mathcal{E}(x)$, if $x$ is a variable
$\mathcal{E}(f(t_1, \ldots, t_N)) = \bigvee_{i=1}^{N} \mathcal{E}(t_i)$

A pair of states $(s = (m, p), t = (n, q))$ satisfies a taint environment $\mathcal{E}$ if $m = n$ (i.e., $s$ and $t$ are at the same program location), and for every variable $x$, if $x$ is untainted in $\mathcal{E}$ (i.e., $\mathcal{E}(x)$ holds), the values of $x$ are equal in $s$ and $t$.

Taint environments are ordered by component-wise implication: $\mathcal{E} \sqsubseteq \mathcal{F} \Leftrightarrow (\forall x : \mathcal{E}(x) \Rightarrow \mathcal{F}(x))$. If $\mathcal{E} \sqsubseteq \mathcal{F}$, then $\mathcal{F}$ taints all variables tainted by $\mathcal{E}$ and maybe more.

### 4.2 Basic Properties

**Proposition 1** *(Monotonicity). If* $(s, t) \models \mathcal{E}$ *and* $\mathcal{E} \sqsubseteq \mathcal{F}$, *then* $(s, t) \models \mathcal{F}$.

For a statement $S$ and states $s = (m, p)$ and $s' = (n, q)$, we write $s \xrightarrow{S} s'$ to mean that there is an execution trace from $s$ to $s'$ such that $m$ denotes the program location immediately before $S$ and $n$ denotes the program location immediately after $S$; $s'$ is the *successor* of $s$ after executing $S$.

In addition, for taint environments $\mathcal{E}$ and $\mathcal{F}$, we write $\{\mathcal{E}\}\, S\, \{\mathcal{F}\}$ to mean that for any pair of states satisfying $\mathcal{E}$, their successors after executing $S$ satisfy $\mathcal{F}$. Formally, $\{\mathcal{E}\}\, S\, \{\mathcal{F}\} \Leftrightarrow (\forall s, t : (s, t) \models \mathcal{E} \wedge s \xrightarrow{S} s' \wedge t \xrightarrow{S} t' : (s', t') \models \mathcal{F})$.

**Proposition 2** *(Widening). If* $\{\mathcal{E}\}\, S\, \{\mathcal{F}\}, \mathcal{E}' \sqsubseteq \mathcal{E}$ *and* $\mathcal{F} \sqsubseteq \mathcal{F}'$, *then* $\{\mathcal{E}'\}\, S\, \{\mathcal{F}'\}$.

### 4.3 Proof System

We present a taint proof system for inferring $\{\mathcal{E}\}\, S\, \{\mathcal{F}\}$ for a structured program $S$. The soundness proof is by induction on program structure, following the pattern of the proof in [17].

$S$ *is* skip: $\{\mathcal{E}\}\ \mathsf{skip}\ \{\mathcal{E}\}$

$S$ *is an assignment* $x\ :=\ e$: $\dfrac{\mathcal{F}(x) = \mathcal{E}(e) \qquad \forall y \neq x : \mathcal{F}(y) = \mathcal{E}(y)}{\{\mathcal{E}\}\, x\ :=\ e\, \{\mathcal{F}\}}$

*Sequence:* $\dfrac{\{\mathcal{E}\}\, S_1\, \{\mathcal{G}\} \qquad \{\mathcal{G}\}\, S_2\, \{\mathcal{F}\}}{\{\mathcal{E}\}\, S_1; S_2\, \{\mathcal{F}\}}$

*Conditional:* For a statement $S$, we use $Assign(S)$ to represent a set of variables which over-approximates those variables assigned to in $S$. The following two cases are used to infer $\{\mathcal{E}\}\, S\, \{\mathcal{F}\}$ for a conditional:

Case A: $\dfrac{\mathcal{E}(c) = \text{false} \quad \{\mathcal{E}\}\, S_1\, \{\mathcal{F}\} \quad \{\mathcal{E}\}\, S_2\, \{\mathcal{F}\}}{\{\mathcal{E}\}\ \mathsf{if}\ c\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}\ \{\mathcal{F}\}}$

Case B: $\dfrac{\begin{array}{c}\mathcal{E}(c) = \text{true} \quad \{\mathcal{E}\}\, S_1\, \{\mathcal{F}\} \quad \{\mathcal{E}\}\, S_2\, \{\mathcal{F}\} \\ \forall x \in Assign(S_1) \cup Assign(S_2) : \mathcal{F}(x) = \text{true}\end{array}}{\{\mathcal{E}\}\ \mathsf{if}\ c\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}\ \{\mathcal{F}\}}$

*While Loop:* $\dfrac{\mathcal{E} \sqsubseteq \mathcal{I} \qquad \{\mathcal{I}\}\ \mathsf{if}\ c\ \mathsf{then}\ S\ \mathsf{else\ skip\ fi}\ \{\mathcal{I}\} \qquad \mathcal{I} \sqsubseteq \mathcal{F}}{\{\mathcal{E}\}\ \mathsf{while}\ c\ \mathsf{do}\ S\ \mathsf{od}\ \{\mathcal{F}\}}$

**Theorem 5** ***(Soundness).*** *Consider a structured program $P$ with a valid proof such that $\{\mathcal{E}\}\, P\, \{\mathcal{F}\}$ holds. For all initial states $(s,t)$ such that $(s,t) \models E$: if there are terminating computations from $s$ and $t$ such that $s \xrightarrow{P} s'$ and $t \xrightarrow{P} t'$ hold, then $(s',t') \models F$.*

The proof system can be turned into an algorithm for calculating taints. The proof rule for each statement other than the while can be read as a monotone forward environment transformer. For while loops, the proof rule requires the construction of an inductive environment, $I$. This can be done through a straightforward least fixpoint calculation for $I$ based on the transformer for the body of the loop. Let $I^k$ denote the value at the $k$'th stage. Each non-final fixpoint step from $I^n$ to $I^{n+1}$ must change the taint status of least one variable from untainted in $I^n$ to tainted in $I^{n+1}$, while leaving all tainted variables in $I^n$ tainted in $I^{n+1}$. Thus, the fixpoint is reached in a number of stages that is bounded by the number of variables. In a nested loop structure, the fixpoint for the inner loop must be evaluated multiple times, but this calculation does not have to be done from scratch; it can be started from the incoming environment $E$, which increases monotonically. The entire process is thus in polynomial time.

## 5 A Secure Dead Store Elimination Transformation

From the results of Sect. 3, checking the security of a program transformation after the fact is computationally difficult. A translation validation approach to security is, therefore, unlikely to be practical. The alternative is to build security into the program transformation. In this section, we describe a dead store elimination procedure built around taint analysis, and prove that it is secure.

1. Compute the control flow graph $G$ for the source program $S$
2. Set each internal variable at the initial location as Untainted, each L-input as Untainted, and each H-input as Tainted
3. Do a taint analysis on $G$
4. Do a liveness analysis on $G$ and obtain the set of dead assignments, DEAD
5. **while** DEAD *is not empty* **do**
   Remove an assignment, $A$, from DEAD, suppose it is "$x := e$"
   Let CURRENT be the set of all assignments to $x$ in $G$ except $A$
   **if** *$A$ is post-dominated by* CURRENT **then** [Case 1]
   Replace $A$ with skip
   Update the taint analysis for $G$
   **else if** *$x$ is Untainted at the location immediately before $A$ and $x$ is Untainted at the final location of $G$* **then** [Case 2]
   Replace $A$ with skip
   **else if** *$x$ is Untainted at the location immediately before $A$ and there is no path from $A$ to* CURRENT *and $A$ post-dominates the entry node* **then** [Case 3]
   Replace $A$ with skip
   **else**
   (* Do nothing *)
   **end**
   **end**
6. Output the result as program $T$

**Fig. 2.** Secure dead store elimination algorithm

The algorithm is shown in Fig. 2. It obtains the set of dead assignments and processes them using taint information to determine which ones are secure to remove. For the algorithm, we suppose that the control-flow graph is in a simplified form where each edge either has a guarded command with a skip action, or a trivial guard with an assignment. I.e., either $g \rightarrow$ skip or $true \rightarrow x := e$. The taint proof system is given for structured programs, so we suppose that the input program is structured, and the CFG obtained from it corresponds to that for a structured program. The "removal" of dead stores is done by replacing the store with a skip, so the CFG structure is unchanged. (The restriction to structured programs is for simplicity and may be relaxed, as discussed in Sect. 6.)

Removal of dead stores can cause previously live stores to become dead, so the algorithm should be repeated until no dead store can be removed. In Case 1 of the algorithm, it is possible for the taint proof to change as well, so the

algorithm repeats the taint analysis. For cases 2 and 3, we establish and use the fact that removal does not alter the taint proof.

As the algorithm removes a subset of the known dead stores, the transformation is correct. In the following, we prove that it is also secure. We separately discuss each of the (independent) cases in the algorithm. For each case, we give an illustrative example followed by a proof that the store removal is secure.

### 5.1 Case 1: Post-domination

The example in Fig. 3 illustrates this case. In the program on the left, two dead assignments to $x$ are redundant from the viewpoint of correctness. Every path to the exit from the first assignment, `x = 0`, passes through the second assignment to $x$. This is a simple example of the situation to which Case 1 applies. The algorithm will remove the first dead assignment, resulting in the program to the right. This is secure as the remaining assignment blocks the password from being leaked outside the function. The correctness of this approach in general is proved in the following lemmas.

```
void foo()
{
  int x;

  x = read_password();
  use(x);
  x = 0;  // Dead Store
  x = 5;  // Dead Store
  return;
}
```

```
void foo()
{
  int x;

  x = read_password();
  use(x);

  x = 5;  // Dead Store
  return;
}
```

**Fig. 3.** C programs illustrating Case 1 of the algorithm

**Lemma 1** *(Trace Correspondence). Suppose that $T$ is obtained from $S$ by eliminating a dead store, $x := e$. For any starting state $s = (H = a, L = c)$, there is a trace in $T$ from $s$ if, and only if, there is a trace in $S$ from $s$. The corresponding traces have identical control flow and, at corresponding points, have identical values for all variables other than $x$, and identical values for $x$ if the last assignment to $x$ is not removed.*

**Proof:** (Sketch) This follows from the correctness of dead store elimination, which can be established by showing that the following relation is a bisimulation. To set up the relation, it is easier to suppose that dead store $x := e$ is removed by replacing it with $x := \perp$, where $\perp$ is an "undefined" value, rather than by replacement with skip. The $\perp$ value serves to record that the value of $x$ is not important. Note that the CFG is unaltered in the transformation. The relation connects states $(m, s)$ of the source and $(n, t)$ of the target if (1) $m = n$ (i.e.,

same CFG nodes); (2) $s(y) = t(y)$ for all $y$ other than $x$; and (3) $s(x) = t(x)$ if $t(x) \neq \perp$. This is a bisimulation (cf. [11], where a slightly weaker relation is shown to be a bisimulation). From this the claim of the corresponding traces having identical control-flow follows immediately, and the data relations follow from conditions (2) and (3) of the relation. **EndProof.**

**Lemma 2.** *If $\alpha$ is a dead assignment to variable $x$ in program $S$ that is post-dominated by other assignments to $x$, it is secure to remove it from $S$.*

**Proof:** Let $T$ be the program obtained from $S$ by removing $\alpha$. We show that any leaky triple for the transformed program $T$ is already present in the source program $S$. Let $(a, b, c)$ be a leaky triple for $T$. Let $\tau_a$ (resp. $\sigma_a$) be the trace in $T$ (resp. $S$) from the initial state $(H = a, L = c)$. Similarly, let $\tau_b$ (resp. $\sigma_b$) be the trace in $T$ (resp. $S$) from $(H = b, L = c)$. By trace correspondence (Lemma 1), $\sigma_a$ and $\sigma_b$ must also reach the exit point and are therefore terminating.

By the hypothesis, the last assignment to $x$ before the exit point in $\sigma_a$ and $\sigma_b$ is not removed. Hence, by Lemma 1, $\tau_a$ and $\sigma_a$ agree on the value of all variables at the exit point, including on the value of $x$. Similarly, $\tau_b$ and $\sigma_b$ agree on all variables at the exit point. As $(a, b, c)$ is a leaky triple for $T$, the $L$-values are different at the final states of $\tau_a$ and $\tau_b$. It follows that the $L$-values are different at the final states for $\sigma_a$ and $\sigma_b$, as well, so $(a, b, c)$ is also a leaky triple for $S$. **EndProof.**

### 5.2 Case 2: Stable Untainted Assignment

An example of this case is given by the programs in Fig. 4. Assume user id to be public and password to be private, hence *read_password*() returns a H-input value while *read_user_id*() returns a L-input value. There are two dead

```
int foo()
{
  int x, y;

  x = 0;  // Dead Store
  y = read_user_id();
  if(is_valid(y)){
   x = read_password();
   log_in(x, y);
   x = 1; // Dead Store
  }else{
   printf("Invalid ID");
  }
  return y;
}
```

```
int foo()
{
  int x, y;


  y = read_user_id();
  if(is_valid(y)){
   x = read_password();
   log_in(x, y);
   x = 1; // Dead Store
  }else{
   printf("Invalid ID");
  }
  return y;
}
```

**Fig. 4.** C programs illustrating Case 2 of the algorithm

assignments to $x$ in the program on the left, and the algorithm will remove the first one, as $x$ is untainted before that assignment and untainted at the final location as well. This is secure as in the program on the right $x$ remains untainted at the final location, and no private information about the password will be leaked via $x$. The general correctness proof is given below.

**Lemma 3.** *Suppose that there is a taint proof for program $S$ where (1) $x$ is untainted at the final location and (2) $x$ is untainted at the location immediately before a dead store, then it is secure to eliminate the dead store.*

**Proof:** The same proof outline is valid for the program $T$ obtained by replacing the dead store "$x := e$ with "skip". Let $\{\mathcal{E}\}\, x := e\, \{\mathcal{F}\}$ be the annotation for the dead store in the proof outline. By the inference rule of assignment, we know that $\mathcal{F}(x) = \mathcal{E}(e)$ and that, for all other variables $y$, $\mathcal{F}(y) = \mathcal{E}(y)$.

Now we show that $\mathcal{E} \sqsubseteq \mathcal{F}$ is true. Consider any variable $z$. If $z$ differs from $x$, then $\mathcal{E}(z) \Rightarrow \mathcal{F}(z)$, as $\mathcal{E}(z) = \mathcal{F}(z)$. If $z$ is $x$, then by hypothesis (2), as $x$ is untainted in $\mathcal{E}$, $\mathcal{E}(z) \Rightarrow \mathcal{F}(z)$ is trivially true, as $\mathcal{E}(z) = \mathcal{E}(x)$ is false.

The annotation $\{\mathcal{E}\}\,\mathsf{skip}\,\{\mathcal{E}\}$ is valid by definition, therefore $\{\mathcal{E}\}\,\mathsf{skip}\,\{\mathcal{F}\}$ is also valid by $\mathcal{E} \sqsubseteq \mathcal{F}$ and Proposition 2. Hence, the replacement of an assignment by skip does not disturb the proof. The only other aspect of the proof that can depend on the eliminated assignment is the proof rule for a conditional (Case B). However, this remains valid as well, as it is acceptable for the set of variables that are forced to be tainted to be an over-approximation of the set of assigned variables.

By hypothesis (1), $x$ is untainted at the final location in $S$. As the proof remains unchanged in $T$, $x$ is untainted at the final location in $T$. By the soundness of taint analysis, there is no leak in $T$ from variable $x$. Hence, any leak in $T$ must come from variable $y$ different from $x$. By trace correspondence (Lemma 1), those values are preserved in the corresponding traces; therefore, so is any leak. **EndProof.**

### 5.3 Case 3: Final Assignment

The example in Fig. 5 illustrates this case. Assume the credit card number to be private, so that `credit_card_no()` returns an H-input value. In the program on the left, there are two dead assignments to $x$. The first one is post-dominated by the second one, while the second one is always the final assignment to x in every terminating computation, and x is untainted before it. By Case 1, the algorithm would remove the first one and keep the second one. Such a transformation is secure, as the source program and result program leaks same private information. But Case 3 of the algorithm would do a better job: it will remove the second dead assignment instead, resulting in the program on the right. We show that the result program is at least as secure as the source program (in this very example, it is actually more secure than the source program), as $x$ becomes untainted at the final location and no private information can be leaked outside the function via $x$. The following lemma proves the correctness of this approach.

```
void foo()
{
  int x, y;

  y = credit_card_no();
  x = y;
  use(x);
  x = 0; // Dead Store
  x = last_4_digits(y); // Dead Store
  y = 0; // Dead Store
  return;
}
```

```
void foo()
{
  int x, y;

  y = credit_card_no();
  x = y;
  use(x);
  x = 0; // Dead Store

  y = 0; // Dead Store
  return;
}
```

**Fig. 5.** C programs illustrating Case 3 of the algorithm

**Lemma 4.** *Suppose that there is a proof outline in the system above for program $S$ where (1) $x$ is untainted at the location immediately before a dead store, (2) no other assignment to $x$ is reachable from the dead store, and (3) the store post-dominates the entry node, then it is secure to eliminate the dead store.*

**Proof:** Similar to Lemma 3, we can prove that the proof outline remains correct for the program $T$ obtained by replacing the dead store "$x := e$" with skip. By hypothesis (1), $x$ is still untainted at the same location in $T$.

By hypothesis (3), the dead store "$x := e$" is a top-level statement, i.e. it cannot be inside a conditional or while loop, thus the dead store (resp. the corresponding skip) occurs only once in every terminating computation of $S$ (resp. $T$). Let $t_a, \ldots, t'_a, \ldots, t''_a$ be the terminating trace in $T$ from the initial state $(H = a, L = c)$, and $t_b, \ldots, t'_b, \ldots, t''_b$ be the terminating trace in $T$ from the initial state $(H = b, L = c)$ where $t'_a$ and $t'_b$ are at the location immediately before the eliminated assignment. By the soundness of taint analysis, $x$ must have identical values in $t'_a$ and $t'_b$.

By hypothesis (2), the value of $x$ is not modified in the trace between $t'_a$ and $t''_a$ (or between $t'_b$ and $t''_b$). Thus, the values of $x$ in $t''_a$ and $t''_b$ are identical, and there is no leak in $T$ from $x$. Hence, any leak in $T$ must come from a variable $y$ different from $x$. By trace correspondence (Lemma 1), those values are preserved in the corresponding traces; therefore, so is any leak. **EndProof.**

**Theorem 6.** *The algorithm for dead store elimination is secure.*

**Proof:** The claim follows immediately from the secure transformation properties shown in Lemmas 2, 3 and 4. **EndProof.**

Although the dead store elimination algorithm is secure, it is sub-optimal in that it may retain more dead stores than necessary. Consider the program "`x = read_password(); use(x); x = read_password(); return;`". The second store to `x` is dead and could be securely removed, but it will be retained by our heuristic procedure.

The case at the end of Sect. 2, in which the transformed program reveals the entire credit card number, cannot happen with dead store elimination. More generally, we conjecture that this algorithm preserves the amount of leaked information. Although there is not a single accepted definition of quantitative leakage, it appears natural to suppose that if two programs have identical computations with identical leaked values (if any) then the amounts should be the same. This is the case in our procedure. By Lemma 1, all variables other than $x$ have identical values at the final location in the corresponding traces of $S$ and $T$. From the proofs of Theorem 6, we know that at the final location of $T$, variable $x$ has either the same value as in $S$ (Case 1) or an untainted value (Cases 2 and 3) that leaks no information, thus $T$ cannot leak more information than $S$.

## 6 Discussion

In this section, we discuss variations on the program and security model and consider the question of the security of other common compiler transformations.

### 6.1 Variations and Extensions of the Program Model

*Unstructured While Programs.* If the while program model is extended with `goto` statements, programs are no longer block-structured and the control-flow graph may be arbitrary. The secure algorithm works with CFGs and is therefore unchanged. An algorithm for taint analysis of arbitrary CFGs appears in [5,6]. This propagates taint from tainted conditionals to blocks that are solely under the influence of that conditional; such blocks can be determined using a graph dominator-based analysis. The full version of this paper contains a taint proof system for CFGs that is based on these ideas. It retains the key properties of the simpler system given here; hence, the algorithms and their correctness proofs apply unchanged to arbitrary CFGs.

*Procedural Programs.* An orthogonal direction is to enhance the programming model with procedures. This requires an extension of the taint proof system to procedures, but that is relatively straightforward: the effect of a procedure is summarized on a generic taint environment for the formal parameters and the summary is applied at each call site. A taint analysis algorithm which provides such a proof must perform a whole-program analysis.

A deeper issue, however, is that a procedure call extends the length of time that a tainted value from the calling procedure remains in memory. Hence, it may be desirable to ensure that all local variables are untainted before a long-running procedure invocation. This can be modeled by representing the location before a procedure invocation as a potential leak location, in addition to the exit point. We believe that the analysis and algorithms developed here can be adapted to handle multiple leak locations; this is part of ongoing work.

### 6.2 The Security of Other Compiler Transformations

Dead store elimination is known to be leaky. In the following, we show that, for several common compiler transformations, security follows from correctness.

The correctness of a transformation from program $S$ to program $T$ is shown using a refinement relation, $R$. For states $u, v$ of a program $P$, define $u =_L v$ ($u$ and $v$ are "low-equivalent") to mean that $u$ and $v$ agree on the values of all Low-variables in program $P$. We say that $R$ is a *strict refinement* if $R$ is a refinement relation from $T$ to $S$ and, in addition:

(a) A final state of $T$ is related by $R$ only to a final state of $S$
(b) If $R(t_0, s_0)$ and $R(t_1, s_1)$ hold, then $t_0 =_L t_1$ (relative to $T$) if, and only if, $s_0 =_L s_1$ (relative to $S$). This condition needs to hold only when $t_0$ and $t_1$ are both initial states or are both final states of $T$.

**Theorem 7.** *Consider a transformation from program $S$ to program $T$ which does not change the set of high variables and is correct through a strict refinement relation $R$. Such a transformation is secure.*

**Proof:** Consider a leaky triple $(a, b, c)$ for $T$. Let $\tau_a$ be the computation of $T$ from initial state $(H = a, L_T = c)$, similarly let $\tau_b$ be the computation of $T$ from initial state $(H = b, L_T = c)$. Let $t_a, t_b$ be the final states of $\tau_a, \tau_b$, respectively. As the transformation is correct, one needs to consider only the case of a leak through the low variables at the final state. By assumption, there is a leak in $T$, so that $t_a =_L t_b$ is false. We show that there is a corresponding leak in $S$.

Let $\sigma_a$ be the computation of $S$ which corresponds to $\tau_a$ through $R$, such a computation exists as $R$ is a refinement relation. Similarly let $\sigma_b$ correspond to $\tau_b$ through $R$. By condition (a) of strictness, the state of $\sigma_a$ ($\sigma_b$) that is related to the final state of $\tau_a$ ($\tau_b$) must be final for $S$, hence, $\sigma_a$ and $\sigma_b$ are terminating computations. Apply condition (b) to the initial states of the corresponding computations $\tau_a, \sigma_a$ and $\tau_b, \sigma_b$. As the initial $\tau$-states are low-equivalent, condition (b) implies that the initial $\sigma$-states are low-equivalent. Applying condition (b) to the final states of the corresponding computations, and using the assumption that $t_a =_L t_b$ is false, the final $\sigma$-states are *not* low-equivalent. Hence, $(a, b, c)$ is also a leaky triple for $S$. **EndProof.**

Informally, a functional definition of the refinement relation ensures condition (b) of strictness. Precisely, we say that a refinement relation $R$ is functional if:

(a) Every low state variable $x$ of $S$ has an associated *1-1* function $f_x(Y_x)$, where $Y_x = (y_1, \ldots, y_k)$ is a vector of low state variables of $T$. We say that each $y_i$ in $Y_x$ *influences* $x$.
(b) Every low state variable $z$ of $T$ influences some low-state variable of $S$
(c) For every pair of states $(t, s)$ related by $R$, $s(x)$ equals $f_x(t(y_1), \ldots, t(y_k))$

**Lemma 5.** *A functional refinement relation satisfies condition (b) of strictness.*

**Proof:** Suppose that $R(t_0, s_0)$ and $R(t_1, s_1)$ hold. By conditions (a) and (c) of the assumption, for every low state variable $x$ of $S$, $s_0(x)$ equals $f_x(t_0(Y_x))$ and $s_1(x)$ equals $f_x(t_1(Y_x))$.

First, suppose that $t_0 =_L t_1$. As $t_0$ and $t_1$ agree on the values of all low variables in $Y_x$, $s_0(x)$ and $s_1(x)$ are equal. This holds for all $x$, so that $s_0 =_L s_1$. Next, suppose that $t_0 =_L t_1$ does not hold. Hence, $t_0(y) \neq t_1(y)$ for some low state variable $y$ of $T$. By condition (b) of the assumption, $y$ influences some low-state variable of $S$, say $x$. I.e., $y$ is a component of the vector $Y_x$ in the function $f_x(Y_x)$. Hence, $t_0(Y_x)$ and $t_1(Y_x)$ are unequal vectors. Since $f_x$ is 1-1, it follows that $s_0(x) = f_x(t_0(Y_x))$ and $s_1(x) = f_x(t_1(Y_x))$ differ, so that $s_0 =_L s_1$ does not hold. **EndProof.**

The refinement relations for a number of transformations are defined functionally. For instance, constant propagation and control-flow simplifications do not alter the set of variables, and the refinement relation equates the values of each variable $x$ in corresponding states of $S$ and $T$. Hence, the relation meets the conditions of Lemma 5 and, therefore, condition (b) of strictness. These relations also satisfy condition (a), as the transformations do not change the termination behavior of the source program.

Dead-store removal *does not* does not have a functionally defined refinement relation. In the example from Fig. 1, the final value of $x$ in the source (which is 0) cannot be expressed as a function of the final value of $x$ in the result (which is arbitrary).

### 6.3 Insecurity of SSA

Another important transformation which *does not* meet the functionality assumption is the single static assignment (SSA) transformation. For instance, consider this transformation applied to the example of Fig. 1. In the program on the right of Fig. 6, the assignments to `x` have been replaced with single assignments to `x1` and to `x2`. The final value of `x` in the source is the final value of `x2`; however, `x1` does not influence `x` at all, violating condition (b) of functionality.

```
void foo()
{
  int x;

  x = read_password();
  use(x);
  x = 0; // clear password
  return;
}
```

```
void foo()
{
  int x1,x2;

  x1 = read_password();
  use(x1);
  x2 = 0;
  return;
}
```

**Fig. 6.** `C` programs illustrating the insecurity of SSA transformation

As SSA transformation is crucial to the operation of modern compilers, the potential for a leak is particularly troubling. One possible resolution is to pass

on taint information to the register allocation stage, forcing the allocator to add instructions to clear the memory reserved for `x1`, unless the memory has been reallocated subsequently to an untainted variable. Investigation of such remedies is a topic for future work.

## 7 Related Work and Conclusions

The fact that correctness preservation is not the same as security preservation has long been known. Formally, the issue is that refinement in the standard sense, as applied for correctness, does not preserve security properties. Specifically, a low-level machine model may break security guarantees that are proved on a higher-level language model. Full abstraction has been proposed as a mechanism for preserving security guarantees across machine models in [1]. A transformation $\tau$ is fully abstract if programs $P$ and $Q$ are indistinguishable (to an attacker context) at level $L_1$ if and only if the transformed programs $P' = \tau(P)$ and $Q' = \tau(Q)$ are indistinguishable at level $L_2$. Recent work on this topic [2,8,14] considers various mechanisms for ensuring full abstraction. This work relates to the preservation of security across machine levels, while our work relates to the preservation of security within a single level. For a single level, one can show full abstraction by proving that $P$ and $\tau(P)$ are indistinguishable. That is essentially the method followed in this paper.

The earliest explicit reference to the insecurity of dead store elimination that we are aware of is [10], but this issue has possibly been known for a longer period of time. Nevertheless, we are not aware of other constructions of a secure dead store elimination transformation. The complexity results in this paper on the difficulty of translation validation for security, in particular for the apparently simple case of dead store elimination, are also new to the best of our knowledge.

Theorem 7 in Sect. 6, which shows that strong refinement relations do preserve security is related to Theorem 10.5 in [2] which has a similar conclusion in a different formal setting. The new aspect in this paper is the application of Theorem 7 to reduce the security of several common compiler transformations to their correctness.

In a recent paper [7], the authors carry out a detailed study of possible ways in which compiler transformations can create information leaks. The authors point out that the correctness-security gap (their term) can be understood in terms of observables: establishing security requires more information about internal state to be observable than that needed to establish correctness. (This is essentially the full abstraction property discussed above.) They describe several potential approaches to detecting security violations. The inherent difficulty of security checking has implications for translation validation and testing, two of the approaches considered in [7]. Our secure dead code elimination algorithm removes an important source of insecurity, while Theorem 7 reduces the security of several other transformations to establishing their correctness with strong refinement relations.

There is a considerable literature on type systems, static analyses and other methods for establishing (or testing) the security of a *single* program, which we

will not attempt to survey here. In contrast, this paper treats the *relative security* question: is the program resulting from a transformation at least as secure as the original? This has been less studied, and it has proved to be an unexpectedly challenging question. Several new directions arise from these results. Securing the SSA transformation is an important concern, as is understanding the security of other common compiler transformations. A witnessing structure for security, analogous to the one for correctness in [11], might be a practical way to formally prove the security of compiler implementations. A different direction is to consider transformations that enhance security, rather than just preserve it; one such transformation is described in [9]. The ultimate goal is a compilation process that is both correct and secure.

**Acknowledgements.** We would like to thank Lenore Zuck, V.N. Venkatakrishnan and Sanjiva Prasad for helpful discussions and comments on this research. This work was supported, in part, by DARPA under agreement number FA8750-12-C-0166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

## References

1. Abadi, M.: Protection in programming-language translations. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 868–883. Springer, Heidelberg (1998)
2. de Amorim, A.A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 165–178. ACM (2014). http://doi.acm.org/10.1145/2535838.2535839
3. Bell, D., LaPadula, L.: Secure computer systems: Mathematical foundations, vol. 1-III. Technical report ESD-TR-73-278, The MITRE Corporation (1973)
4. Ceara, D., Mounier, L., Potet, M.: Taint dependency sequences: a characterization of insecure execution paths based on input-sensitive cause sequences. In: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, 7–9 April 2010. Workshops Proceedings, pp. 371–380 (2010). http://dx.doi.org/10.1109/ICSTW.2010.28
5. Denning, D.E.: Secure information flow in computer systems. Ph.D. thesis, Purdue University, May 1975
6. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977). http://doi.acm.org/10.1145/359636.359712
7. D'Silva, V., Payer, M., Song, D.X.: The correctness-security gap in compiler optimization. In: 2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, 21–22 May 2015, pp. 73–87. IEEE Computer Society (2015). http://dx.doi.org/10.1109/SPW.2015.33

8. Fournet, C., Swamy, N., Chen, J., Dagand, P., Strub, P., Livshits, B.: Fully abstract compilation to JavaScript. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, Rome, Italy, 23–25 January 2013, pp. 371–384. ACM (2013). http://doi.acm.org/10.1145/2429069.2429114
9. Gondi, K., Bisht, P., Venkatachari, P., Sistla, A.P., Venkatakrishnan, V.N.: SWIPE: eager erasure of sensitive data in large scale systems software. In: Bertino, E., Sandhu, R.S. (eds.) Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, 7–9 February 2012, pp. 295–306. ACM (2012). http://doi.acm.org/10.1145/2133601.2133638
10. Howard, M.: When scrubbing secrets in memory doesn't work (2002). http://archive.cert.uni-stuttgart.de/bugtraq/2002/11/msg00046.html. Also https://cwe.mitre.org/data/definitions/14.html
11. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 304–323. Springer, Heidelberg (2013)
12. Necula, G.: Translation validation of an optimizing compiler. In: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI 2000), pp. 83–95 (2000)
13. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1994)
14. Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., Piessens, F.: Secure compilation to protected module architectures. ACM Trans. Program. Lang. Syst. **37**(2), 6 (2015). http://doi.acm.org/10.1145/2699503
15. Pnueli, A., Shtrichman, O., Siegel, M.: The code validation tool (CVT)- automatic verification of a compilation process. Softw. Tools Technol. Transf. **2**(2), 192–201 (1998)
16. Smith, G.: Recent developments in quantitative information flow (invited tutorial). In: 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, 6–10 July 2015, pp. 23–31. IEEE (2015). http://dx.doi.org/10.1109/LICS.2015.13
17. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996). http://dx.doi.org/10.3233/JCS-1996-42-304
18. Zuck, L.D., Pnueli, A., Goldberg, B.: VOC: a methodology for the translation validation of optimizing compilers. J. UCS **9**(3), 223–247 (2003)