Securing The SSA Transform

Chaoqiang Deng[1] and Kedar S. Namjoshi[2]

[1] New York University `deng@cs.nyu.edu`
[2] Bell Laboratories, Nokia `kedar.namjoshi@nokia-bell-labs.com`

Abstract. Modern optimizing compilers use the single static assignment (SSA) format for programs, as it simplifies program analysis and transformation. A source program is converted to an equivalent SSA form before it is optimized. The conversion may, however, create a less secure program if fresh SSA variables inadvertently leak sensitive values that are masked in the original program. This work defines a mechanism to restore a program to its original security level after it has been converted to SSA form and modified further by a series of optimizing transformations. The final program is converted out of SSA form by grouping variables together in a manner that blocks any new leak introduced by the initial SSA conversion. The grouping relies on taint and leakage information about the original program, which is propagated through the optimizing transformations using refinement proofs.

1 Introduction

A compiler carries out a transformation of a high level, abstract program to low level executable code. Ensuring that compilation preserves program semantics is, therefore, a critical question. It has received much attention, both from the point of view of detecting errors in existing compilers (cf. [20,11] and related work) and from the viewpoint of formally guaranteeing correct compilation (cf. [15,12] and related work). In today's world, it is important to also guarantee that compilation does not weaken security properties. For example, compiler transformations should not inadvertently introduce new pathways that leak sensitive data. It is this issue that is considered in this paper.

Correctness and security turn out to be distinct issues. A well-known illustration is the dead-store elimination optimization, which removes a store instruction from a program if the value stored is never used. This has an unfortunate consequence that removal of stores may introduce an information leak. A simple example is the program shown on the left in Figure 1. It reads a password into a variable x, checks it for validity, and subsequently clears the secret from x by setting it to 0. As the value 0 stored to x is never used, this store is removed (silently) by the dead-store optimization. That leaves the password on the stack or in a register longer than was originally intended, making it possible for an attack elsewhere in the program to obtain the password. The optimization is correct in that it preserves the input-output behavior of the original program; however, it is insecure.

In recent work [6], we formulate a notion of a secure program transformation in terms of information flow (precisely, non-interference as in [7]) and show that checking the security of a dead-store elimination after the fact is undecidable in general. We also define a method which limits the removed dead stores to those where removal provably preserves security. In that paper it is also shown that the static single assignment (SSA) transformation is insecure. The technique used for securing dead-store elimination unfortunately does not apply to SSA. The question of how to secure SSA was left open. In this work, we present a mechanism to secure the SSA transformation.

```
void foo()
{
  int x;

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

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

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

Fig. 1. `C` program illustrating the insecurity of SSA transformation, from [6]

An example of how SSA may cause an information leak is shown in Figure 1, taken from [6]. The source program is the password-reading program described above. The SSA transform introduces fresh names `x1` and `x2` for the assignments to `x`. Suppose that `x1` and `x2` are assigned distinct registers. The assignment to `x2` then has no effect and at the call to function `other`, the password is in the clear, stored in the register assigned to `x1`. Suppose further that there is a vulnerability in the function `other` by which an attacker can gain control of the program. The attacker can then read off the password, which is either in the register assigned to `x1`, or is stored on the call stack. This is a new information leak, one that is not present in the original program.

The leak can be prevented if `x1` and `x2` are always allocated the same register which cannot, however, be guaranteed. Moreover, it is inefficient and not always correct to forcibly clear any tainted data before an untrusted function call. This is because a sound taint analysis is generally over-approximate: variables that are declared tainted may, in fact, always contain either non-sensitive data, or sensitive data that is used after the function call.

Our method, therefore, does not modify the program in any essential manner. The unSSA transform groups together related SSA variables and renames every variable in a group G to a single fresh variable, say z_G. A register allocator is thus forced to assign a single register to the group (subject to live range splitting, which we discuss at the end of the paper). For the example program, `x1` and x2 are placed in the same group so the renaming, in effect, restores the original program.

As grouping and renaming destroys the single assignment property of SSA, this transformation must be placed in the compiler only after all SSA-dependent transformations have been performed. This introduces a key problem. A modern compiler first converts a source program P to its SSA form, say Q_0, then applies a series of SSA-to-SSA optimizations, which result in equivalent but syntactically different intermediate programs $Q_1, \ldots, Q_n$. The unSSA conversion must, therefore, be applied to the final program Q_n, but block leaks introduced in the initial conversion to Q_0. Doing so correctly requires preserving leakage-relevant information through the series of SSA optimizations, which is done with the help of refinement proofs for each transformation. The unSSA transform uses the preserved information to determine a proper grouping of variables in Q_n. Although it is helpful for register allocation to produce small groups, we show that finding the optimal partitioning is undecidable. The unSSA transformation thus relies on sound but approximate analysis information to define the grouping.

The unSSA transform converts the SSA program Q_n to a non-SSA program R; the executable program X is then created by register allocation on R. The overall correctness claim shows that the object code X is at least as secure as the original program P. This is a *relative security* claim: one obtains only that any information leak in X has a corresponding leak in P, *not* that X is free of information leaks. It is analogous to the claim of compiler correctness, which also shows only that every input-output behavior of X can be found in P; not that X is correct in an absolute sense with respect to a specification.

To summarize, the paper studies the important question of making certain that a compiler transformation does not introduce new information leaks. We show that fresh leaks introduced by conversion to SSA form can be blocked by a suitable unSSA transformation, which groups SSA variables into equivalence classes. We show that finding the optimal grouping is, unfortunately, undecidable; hence, our algorithm relies on approximate taint and control-flow information. We also show that leaks may be introduced during register allocation, in particular during live-range splitting, and suggest a simple mechanism to block such a leak. The result is a secure end-to-end compilation.

2 Background

This section contains background on information leakage and taint analysis. Several basic definitions are taken from [6].

Program Syntax and Semantics. As illustrated in the introductory example, the SSA transformation leaks information when sensitive data is retained in registers that can be accessed in a nested procedure call. We can thus consider the point of invocation of an untrusted procedure as a potential *leak point* and require that any leak through a leak point in the final program is a leak that occurs in the source. To simplify the formal development and bring out the key features of the proposed method, the formal development is in terms of structured WHILE programs over integer variables, defined by the syntax below. The program operates on a set of input and state variables. Input variables are partitioned into H ("high security") and L ("low security") variables. All state variables are considered to be low security variables.

$x \in \mathbb{X}$ 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 $\mathbb{X}$
$S \in \mathbb{S} ::= \mathsf{skip} \mid \mathsf{out}(e) \mid x := e \mid ||i : v_i := \phi(v_{i,A}, v_{i,B}) \mid S_1; S_2 \mid$
$\mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid \mathsf{while}\ g\ \mathsf{do}\ S\ \mathsf{od}$ statements

A program can be represented by its *control flow graph* (CFG). (The conversion is standard, and omitted.) A node of the CFG represents a program location, and an edge is labeled with a guarded command, of the form "$g \to a$", where g is a Boolean predicate and a is a primitive statement, one of skip, out (output), or assignment. 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.

Dominance. Node n of a CFG *post-dominates* node m if all paths in the CFG from m to the exit node pass through n.

Program Semantics. The semantics of a program is defined in the standard manner. A *program state* s is a triple (m, e, p), where m is a CFG node, referred to as the *location* of s; if m is not the entry node, then e is the entry edge, (k, m) for some k, and p is a function that maps each variable to an integer value. The function p can be extended to evaluate an expression in the standard way (omitted). An *initial* state has the form $(\mathsf{entry}, \perp, p)$ where $p(x) = 0$ for all state variables x. A *final* state has the form (exit, e, p).

A pair of states, $(s = (m, e, p), t = (n, f, q))$ is in the *transition relation* if $f = (m, n)$ is an edge of the CFG, and for the guarded command $g \to a$ on that edge, the guard g holds of p, and the function $q(y)$ is identical to $p(y)$ for all variables y other than those modified by the statement a, for which it is defined as follows. If a is an assignment $x := e$, the only variable modified is x and $q(x)$ equals $p(e)$. For the assignment $||i : v_i := \phi(v_{i,A}, v_{i,B})$, the variables modified are the v_i's, and $q(v_i)$ is given by $p(v_{i,A})$, if $e = A$, and by $p(v_{i,B})$, if $e = B$. For skip and out statements, $q = p$.

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.

Information Leakage. Information leakage is defined using the standard concept of *non-interference* (cf. [3,7]). 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 but 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.

Taint Proofs. Leakage is approximated by the notion of taint. Each variable in a program is marked as either "tainted" or "untainted". High inputs are always tainted, low inputs are always untainted. The correctness of these markings is expressed as a taint proof. Each program point is decorated with a taint assertion E, a function from variables to taint values. In earlier work [6], we define a sound taint proof system in the style of the Volpano-Irvine-Smith proof system [19], which sets up consistency conditions on these taint assertions. The soundness of the proof system implies that a leak cannot occur through any untainted variable. On the other hand, the fact that a variable is tainted does not necessarily imply that there is a leak through that variable.

Correct Transformation. For simplicity, we only 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 execution of Q on a has a corresponding execution of P on a with an identical sequence of output values. Termination is considered an output, so that a terminating execution of Q must have a matching terminating execution of P. A correct transformation thus offers the relative correctness guarantee that every input-output behavior of Q is present in P. It does not assure the correctness of either program with respect to a specification.

Secure Transformation. A transformation is *secure* if the set of leaky triples for Q is a subset of the leaky triples for P. A secure transformation ensures relative security, i.e., that Q is not more leaky than P. It does not ensure that either P or Q are free of information leaks. Suppose that a transformation from P to Q is correct. For any leaky triple (a, b, c) for Q, if the computations of Q from inputs $(H = a, L = c)$ and $(H = b, L = c)$ differ in their output, this difference must also (by correctness) appear in the corresponding computations in P. Hence, the only way in which Q can be less secure than P is if the computations on inputs $(H = a, L = c)$ and $(H = b, L = c)$ terminate in Q with different L-values, but the corresponding P-computations (which must terminate, too, by correctness) have identical L-values.

The SSA Transformation. A program P is converted to its SSA form Q essentially by replacing each assignment to a variable x with a fresh name, say x_i for the i'th assignment. ϕ-functions are inserted at merge points to combine values that reach that point from different branches of a conditional or while statement. Not all merge points need a ϕ function; efficient algorithms to determine the optimal placement of ϕ functions are given in [5,18]. To illustrate the SSA transform by an example, consider the programs shown in Figure 2. The program on the left is transformed to the program on the right. In the process, variable x is given three versions: $x1$, $x2$ and $x3$, and ϕ-functions are inserted at the start of the loop to merge the values of $x1, x3$ into $x2$, and of $i1, i3$ into $i2$.

```
for (i=0, x=0; i < N; i++)
{
  x = x+i;
}
```

```
i1 = 0;
x1 = 0;
loop:
     i2 = phi(i1,i3),
  || x2 = phi(x1,x3);
  if (i2 >= N) goto end;
  x3 = x2 + i2;
  i3 = i2 + 1;
  goto loop;
end:
```

Fig. 2. Illustrating the SSA transformation.

3 Securing SSA

The problem addressed in this work is stated precisely as follows. A program P is converted by a compiler to its SSA form, Q_0, which is transformed by a series of SSA-to-SSA optimizations through intermediate programs $Q_1, \ldots, Q_n$. The conversion from P to Q_0 may introduce new leaks, as illustrated in the introduction. The goal is to block these new leaks via a new, final transform, called the "unSSA" transform, which converts Q_n into a program R that is at least as secure as P. We use this naming convention throughout the section.

3.1 An Overview

The essential idea behind the unSSA transform can be understood by considering the changes that the SSA transformation makes in converting P to Q_0. A variable x of P is represented by several versions, say $x_1, x_2, \ldots, x_k$ in Q_0. Programs P and Q_0 have identical control-flow graphs and are observationally equivalent: the bisimulation relation is that at each common program point p, the value of a variable x at point p in P equals the value of one of its versions, x_i, at point p in Q, where the index i is a function of the location p.

Each version of x in Q_0 represents an assignment to x in P. This exposes all intermediate values of x and is the source of the new leaks, as illustrated in the introduction. In order to block such leaks, one has to reverse the SSA transformation, and assign multiple variants of x the same name. This is done by partitioning the variants of x into groups and rewriting the name of every variable in a group, say G, to a fresh name, say z_G.

The grouping must, however, be done carefully. One option is to group all variants of x into a single class, in which case the result is a program isomorphic to the original P. But that negates an important advantage of the SSA conversion. The conversion naturally splits the live range of a variable x (the extent to which it is live, or used) into disjoint sub-ranges for $x_1, x_2, \ldots, x_k$, reducing interference and improving register allocation. Ideally, one would like to choose a partition that is as fine as possible but no finer: i.e., it should maximize the number of independent groups. This, unfortunately, is undecidable in general.

Theorem 1 *It is undecidable to determine whether two instances of a variable must be grouped together.*

Proof: Consider an arbitrary program P. For a fresh high-security input h and a fresh low security state variable l, define the program $Q(h)$ as P; $l := h$; $l := 0$. Its SSA form is $\tilde{P}$; $l_1 := h$; $l_2 := 0$, where $\tilde{P}$ is the SSA form of P. The SSA transformation of Q leaks the value of h if, and only if, P terminates. Hence $\{l_1, l_2\}$ must be grouped together if, and only if, P terminates. **EndProof.**

The naming scheme followed in Q_0 holds clues to the origin of a variable: for instance, x_8 is a variant of the original variable x. This facilitates grouping: x_8 is never to be grouped with, say, y_3. Subsequent transformations, may, however, alter the set of variables and modify control flow, making it difficult to recover the origin of a variable. If, for example, variables x_8 and y_3 are renamed to u, v, respectively, it is no longer clear from the names that u and v should not be in the same group. A key problem that must be resolved, therefore, is the discovery of the relationships between the variables of the final program Q_n and the variables of Q_0, so that the grouping in Q_n can be done correctly. Our solution is to utilize the refinement relations that connect successive programs Q_i and Q_{i+1}. These relations implicitly hold information that relate variables across transformations. The sketch of the overall procedure is as follows.

– Each transformation from Q_i to Q_{i+1} is witnessed by a refinement relation, ξ_i. From this, one identifies a "core" set of variables, C_i, for each Q_i. The core set is defined so that any leak through the core variables of Q_{i+1} induces a leak via the core variables of Q_i.
  The core set C_0 for Q_0 is defined by the bisimulation relation between P and Q_0. For each variable x of P, its variant x_i is in the core set C_0 if x_i is related to x by the bisimulation at the exit node. (In the introductory example, the core set is $\{x_2\}$.) By transitivity, a leak via the final core set C_n induces a leak in P.
– The variables in Q_n are partitioned into groups, each with a representative variable that is either a core variable, or an untainted non-core variable. The unSSA transform converts Q_n to R by renaming all occurrences of a variable in a group G to the name of the representative of G.
  Additional properties are required to ensure the existence of such a partitioning, and to ensure the correctness of renaming. Those properties, labeled

(P1)-(P3) below, hold for program Q_0 and must be preserved by each transformation.

The constructions guarantee that correctness is preserved when passing from P to R. They also ensure end-to-end security: a leak in R translates to a leak via the core variables of Q_n, which translates to a leak in P. The requirement that properties (P1)-(P3) are preserved across transformations potentially constraints the set of transformations that can be applied. We show that several common transformations do meet those conditions. The rest of this section defines these concepts and properties and gives a proof of the claimed result.

3.2 Core Sets

A "core" set of Q_{i+1} relative to Q_i is a subset X of the variables of Q_{i+1} such that if two end-states of Q_{i+1} differ in the value of some variable in X, the corresponding low end-states of Q_i differ in the value of some variable in the core set C_i of Q_i. Given the core set C_i of Q_i, a set X is a core set of Q_{i+1} if it meets the following constraint.

$$[\xi_i(t,s) \ \wedge\ \mathsf{final}_{i+1}(t) \ \wedge\ \xi_i(t',s') \ \wedge\ \mathsf{final}_{i+1}(t') \ \wedge\ s =_{C_i} s' \ \Rightarrow\ t =_X t'] \qquad (1)$$

In the formulation, the predicate $\mathsf{final}_i(t)$ holds of a state if its location is the exit node of program Q_i; the relation ξ_i is a refinement relation from Q_{i+1} to Q_i; and the relation $=_Y$ represents equality of states on the set Y of variables. I.e., $t =_X t'$ is short for $t[x] = t'[x]$, for all $x \in X$. Informally, the constraint says that a leak in Q_{i+1} that is witnessed by end-states t, t' which differ on some variable in X has a corresponding leak in Q_i, with end states s, s' that differ in some variable of C_i.

It is easy to see from the constraint that if X and Y are core sets, so is $X \cup Y$; and if X is a core set and Y a subset of X, then Y is a core set. By closure under union, there is a largest core set. By closure under subset, the largest core set can be calculated as the set of variables u of Q_{i+1} for which the constraint holds with $X = \{u\}$. Thus, the largest core set of Q_{i+1} relative to C_i, denoted C_{i+1}, can be determined with a linear number of validity checks, one for each variable in Q_{i+1}.

The core set C_0 is defined in Section 3.1. Starting with C_0, one can successively determine the core sets $C_1, C_2, \ldots, C_n$ using the construction procedure above for the programs $Q_1, Q_2, \ldots, Q_n$. The main consequence of the core-set definition is the following theorem.

Lemma 1 *For every i, a leak in Q_{i+1} via its core set C_{i+1} has a corresponding leak in Q_i via its core set C_i.*

Proof: Consider a leaky triple (a, b, c) for Q_{i+1}. Let the induced computations on the inputs $(H = a, L = c)$ and $(H = b, L = c)$ be σ_a and σ_b. As the leak is through C_{i+1}, the final states of these computations, say t_a and t_b, differ for some variable in C_{i+1}. By the refinement relation ξ_i, there are corresponding computations δ_a and δ_b of Q_i whose final states, s_a and s_b, are related to t_a and t_b, respectively, by ξ_i. From implication (1), s_a and s_b differ in C_i; thus, there is a leak in Q_i via C_i. **EndProof.**

Theorem 2 *Any leak in Q_n via its core set C_n has a corresponding leak in P.*

Proof: Using Lemma 1, by induction on the length of the Q sequence, one obtains that a leak in Q_n via C_n corresponds to a leak in Q_0 via C_0. By the definition of C_0, this induces a leak in P. **EndProof.**

3.3 Grouping Variables

While Theorem 2 ensures that a leak through a core set is reflected as a leak in P, it does not cover leaks via non-core variables in Q_n. (In the introductory example, the variable x_1 through which the password is leaked is a non-core variable.) To stop such leaks, the unSSA transform partitions variables of Q_n into groups. Each variable is given the name of its representative, which is either a core variable or an untainted non-core variable. The grouping must be such that (1) it preserves correctness; and (2) it is as lax as possible, i.e., the number of groups is large, to give the register allocator more freedom.

We satisfy these requirements by requiring the following property: for each program Q_i, there is a taint proof and a partition of its variables into classes such that

- (P1) Each class of the partition has a representative variable, which is either a core variable or an untainted non-core variable
- (P2) The variables in a class have mutually disjoint live ranges, assuming that the core variables C_i are live at the end point (Variables with mutually disjoint live-ranges are said to be *interference-free*.)
- (P3) The definition of the representative variable in a class post-dominates the live range of any other variable in its class

These properties hold of the core set C_0 of the initial SSA program, Q_0. The partition has a class for each variable x of the original program P. The class associated with x holds all variants $x_1, \ldots, x_k$ of x in Q_0. The representative of each class is the variant that is in the core set C_0, which satisfies (P1). Property (P2) holds as the variants of x have disjoint live ranges by definition. Property (P3) holds as the members of C_0 are chosen based on the bisimulation between P and Q_0 at the final states.

We show that these properties, applied to Q_n, suffice to define a correct, secure unSSA transform. We then give sufficient conditions which ensure that a transformation preserves (P1)-(P3). Since the properties hold of Q_0, any series of transformations which preserve those properties results in a Q_n to which the unSSA transformation can be applied.

3.4 The unSSA Transform

The unSSA transformation from Q_n to R operates as follows:

1. Construct the core set C_n
2. Perform a taint analysis of Q_n
3. Partition the variables of Q_n into groups such that properties (P1)-(P3) defined above are satisfied
4. For each class Y, create a fresh variable, r_Y, and rename each occurrence of a variable in Y with r_Y

The result is the program R.

Theorem 3 *The unSSA transformation is correct.*

Proof Sketch: The variables in each class are interference-free by (P2). Renaming the variables in class Y with a fresh variable, r_Y, is analogous to the allocation of a physical register to a group of interference-free variables. Correctness thus follows from a standard bisimulation property of register allocation (cf. [16]): the control flow graphs of Q_n and R are isomorphic, and at a point p, the value of r_Y in program R equals the value of a specific variable of class Y in Q_n. **EndProof.**

Properties (P1)-(P3) can, in fact, be established through standard coloring algorithms used for register allocation. The interference graph is built from all variables. Fix a set of representative variables, which include all core variables and some untainted non-core variable names. An interference edge between variables x and y indicates that either the live ranges of x and y intersect, or that x is a representative variable that does not dominate y. The representative variables form the set of colors, and are each colored with their own color. The set of variables that are colored with variable v form the group for v. By construction, there are no edges between variables colored v, so those variables have mutually disjoint live ranges and the representative variable dominates all others.

Theorem 4 *Any leak in program R induces a corresponding leak in P.*

Proof:

Let (a, b, c) be a leaky triple for program R. Thus, the computations σ_a and σ_b from respective inputs $(H = a, L = c)$ and $(H = b, L = c)$ differ at their final states in the value of some variable of R, say r_Y. Consider the corresponding computations, δ_a and δ_b, of program Q_n. Suppose r_Y corresponds to a class Y of variables in Q_n. By the bisimulation relation for register allocation described in the proof of Theorem 3, at each point, the value of r_Y on σ_a (σ_b) equals the value of a specific variable in class Y at the corresponding point on δ_a (δ_b). Assume that r_Y corresponds to variable $u \in Y$ at the end node, then the value of u is different at the end of σ_a and σ_a, hence u cannot be untainted. By (P3), u must be the representative variable in Y. By (P1), as u is a representative and tainted, it must be a core variable of Q_n. Therefore, δ_a and δ_b differ in the value of a core variable of Q_n. By Theorem 2, this leak in Q_n via a core variable has a corresponding leak in P. **EndProof.**

3.5 Preserving the Partitioning Properties

The previous theorems assume that a partitioning of Q_n with properties (P1)-(P3) can be established. As shown, these properties hold for the initial SSA program Q_0. We provide sufficient conditions on SSA-to-SSA transformations which preserve these properties, and show in the following section that these conditions hold for common transformations such as constant propagation and loop unrolling.

The conditions require establishing a simulation relation, ν_i, between the CFG's of Q_{i+1} and Q_i that preserve variable definitions and uses, according to a mapping μ from variables of Q_{i+1} and Q_i. Note that this simulation relation is structural, and thus different from the ξ_i relation referred to earlier, which is on the semantics of the programs.

- (C1) If y is a core or untainted non-core variable of Q_i, there is x such that $\mu(x) = y$ and x is a core or untainted non-core variable of Q_{i+1}
- (C2) The simulation relation ν_i from the CFG of Q_{i+1} to the CFG of Q_i preserves variable definitions and uses. I.e., for any variable x of Q_{i+1}, if x is defined on an edge (n, n') and n is simulated by m, then $\mu(x)$ is defined on the simulating edge (m, m'), and similarly for uses of x

Lemma 2 *Consider a transformation from Q_i to Q_{i+1} which satisfies conditions (C1)-(C2). If variables x, y interfere in Q_{i+1}, then $\mu(x), \mu(y)$ interfere in Q_i.*

Proof: We use the fact that both Q_i and Q_{i+1} are in SSA form. For SSA programs, as shown in [10], variables x and y interfere if, and only if, the definition of one of the variables, say x, dominates the definition of y, and x is live at y. Thus, there is a path in the CFG of Q_{i+1} from definition of x through the definition of y to a use of x. By (C2), this has a simulating path in Q_i which preserves defs and uses; thus, the corresponding path in Q_i starts at the definition of $\mu(x)$, passes through the definition of $\mu(y)$ to a use of $\mu(x)$. Hence, $\mu(x)$ and $\mu(y)$ interfere in Q_i. **EndProof.**

Theorem 5 *Consider a transformation from Q_i to Q_{i+1} which satisfies conditions (C1)-(C2). If Q_i satisfies (P1)-(P3), so does Q_{i+1}.*

Proof: Consider variables x, y to be equivalent in Q_{i+1} if $\mu(x)$ and $\mu(y)$ are equivalent in Q_i. For a class C of Q_{i+1}, let D be the class of Q_i that the members of C are mapped to by μ. Let d be the representative of D. By (P1), d is either a core variable or an untainted non-core variable. By (C1), there is a variable c such that $\mu(c) = d$ and c is a core or untainted non-core variable; this variable must be in class C. Pick one such c as the representative of class C. This establishes (P1).

Now consider variables x, y in C. Then $\mu(x), \mu(y)$ are in D and are, therefore, non-interfering. By the converse of Lemma 2, x and y are non-interfering, establishing (P2). Finally, let x be any variable in C other than the representative c. Consider a path σ in the CFG of Q_{i+1} from a use of x to the exit node.

By (C2), there is a simulating path γ from a use of $\mu(x)$ to the exit node. By (P3) for Q_i, γ must pass through an edge e that defines the core variable $\mu(c)$. Hence, the edge corresponding to e on σ defines c. Therefore, every use of x is post-dominated by the definition of c, establishing (P3). **EndProof.**

4 Example Transformations

We give examples of compiler optimizations which preserve properties (P1)-(P3).

4.1 Constant propagation and folding

This transformation replaces expressions that have a constant value with the value. This is illustrated with the programs in Fig. 3. The original program P is in the top left corner, and has a constant secret value. The SSA transform yields program Q_0 that is in the top right, with core set is $\{x2\}$. Constant propagation and folding optimizes Q_0 to Q_1 that is in the bottom left. In Q_1, the core set is unchanged and $x1$ can be grouped with $x2$. After the unSSA transform, the result is program R in the bottom right, where the information leak introduced by SSA from variable $x1$ is revoked.

```
void foo()
{
  int x;

  x = secret;
  use(x);
  x = 0;
}
```

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

  x1 = secret;
  use(x1);
  x2 = 0;
}
```

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

  x1 = secret;
  use(secret);
  x2 = 0;
}
```

```
void foo()
{
  int x2;

  x2 = secret;
  use(secret);
  x2 = 0;
}
```

Fig. 3. C program illustrating constant propagation and folding

In general, consider a program Q_{i+1} that is obtained from Q_i through constant propagation and folding. The control-flow and variables are unchanged in the transformation. Suppose that there is a taint proof and a partition of variables in Q_i that satisfies (P1)-(P3). First, note that the same taint proof carries over to Q_{i+1}. To see this, consider a statement $x := e$ that is replaced with $x := \bar{e}$, where $\bar{e}$ is obtained by replacing variables in e with their constant values, and folding any constant expressions. Then a taint triple $\{E\}x := e\{F\}$ is valid in Q_i if $E(e) \sqsubseteq F(x)$ and $E(y) \sqsubseteq F(y)$, for all other variables y. As replacing variables with constants can only lead to a stronger taint value, it is the case that $E(\bar{e}) \sqsubseteq E(e)$; hence, $E(\bar{e}) \sqsubseteq F(x)$ by transitivity. Let the function μ be the identity function. As the set of core variables and the taint proof is identical in both programs, condition (C1) holds. Let ν be the identity relation on the control-flow graph nodes; then ν is a simulation which respects uses and defs of variables, establishing (C2). By Theorem 5, conditions (P1)-(P3) are preserved across the transformation.

4.2 Loop unrolling

Loop unrolling is a transformation that replicates the body of a loop. We consider the basic case in Fig. 4, where the original loop is executed an even number of times, and the unrolling factor is 2. The program in the top-left is the original program P, and the program in the top-right is the SSA transformed program Q_0 whose core set is $\{i2, x2\}$. After loop unrolling, the program Q_1 in the bottom-left is generated, with the same core set. In the course of unrolling the body of the loop, all statements are duplicated and the phi-assignment and the test of the loop condition are simplified. The variable partition groups all variants of i together, and all variants of x together, as the corresponding live ranges are disjoint. Renaming the variables in each group and simplifying the resulting statements, one obtains the program R in the bottom-right, which is as secure as the original program P.

Loop unrolling preserves the set of variables and the core set, introducing new variables only inside the loop being unrolled. Suppose that there is a taint proof and a partitioning that satisfies properties (P1)-(P3) for program Q_i. Then the taint proof carries over essentially unchanged to the unrolled loop. In place of the original taint invariant I for the loop, one has the extended taint invariant I', where $I'(x) = I(x)$ for any original variable x, and $I'(x') = I(x)$, for the copy x' of variable x which is introduced in the unrolling. Other taint assertions are unchanged. Let μ be the function defined by $\mu(x) = x$, if x is an original variable, and $\mu(x') = x$, if x' is the copy of x introduced in the unrolling. As core variables and taints are unaffected by the transformation, condition (C1) holds with this definition of μ. Let ν be the relation which connects control-flow nodes in the copy of the loop body with their original nodes. One may verify that ν is a simulation which preserves defs and uses according to μ, establishing (C2). In the example above, `x4` is a copy of `x2` so $\mu(x4) = x2$. The transition defining `x4` in Q_{i+1} is matched by the transition defining `x2` in Q_i; similarly, the definition of `x5` is matched by the transition defining `x3`. By Theorem 5, conditions (P1)-(P3) are preserved across the loop unrolling transformation.

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

  x = 0;
  for(i = 0; i < 2*K; i++)
  {
    x = f(x,i);
  }

  return;
}
```

```
void foo()
{
  int i1,i2,i3,
      x1,x2,x3;

  x1 = 0;
  i1 = 0;
  loop:
    x2 = phi(x1, x3) ||
    i2 = phi(i1, i3);
    if(i2 >= 2*K) goto end;
    x3 = f(x2,i2);
    i3 = i2 + 1;
    goto loop;
  end:

  return;
}
```

```
void foo()
{
  int i1,i2,i3,i4,i5,
      x1,x2,x3,x4,x5;

  x1 = 0;
  i1 = 0;
  loop:
    x2 = phi(x1, x5) ||
    i2 = phi(i1, i5);
    if(i2 >= 2*K) goto end;
    x3 = f(x2,i2);
    i3 = i2 + 1;
    x4 = x3 ||  // phi
    i4 = i3;
    skip;       // test
    x5 = f(x4,i4);
    i5 = i4 + 1;
    goto loop;
  end:

  return;
}
```

```
void foo()
{
  int i2, x2;

  x2 = 0;
  i2 = 0;
  loop:
    if(i2 >= 2*K) goto end;
    x2 = f(x2,i2);
    i2 = i2 + 1;
    x2 = f(x2,i2);
    i2 = i2 + 1;
    goto loop;
  end:

  return;
}
```

Fig. 4. `C` program illustrating loop unrolling

4.3 Secure DSE

The transformations considered so far preserve (P1)-(P3) but do not make use of the properties in defining the transformation itself. This section proposes a novel dead store elimination (DSE) transformation for SSA programs which does both. Similar to the DSE transformation introduced in [6], this is a heuristic to determine whether it is secure to remove a certain dead store. To be precise, for a SSA program Q_i satisfying partitioning properties (P1)-(P3), the transformation removes a dead assignment "$x := e$" is removed if either the partition containing x has size 1, or x is not a representative variable for its partition.

Lemma 3 *The single-step DSE transformation preserves (P1)-(P3).*

Proof: Consider a taint proof for program Q_i. Let E be the taint assertion just before the dead assignment to x. As Q_i is in SSA form, this is the only assignment to x. As it is dead, x is never referenced in Q_i. At all points in Q_{i+1}, let the taint status of x be "untainted". As x is never referenced in Q_i, this change does not affect the taint status of any other variable. As x is never modified in Q_{i+1}, it is correct to assert that x is untainted throughout.

Suppose the variables in Q_i are partitioned into classes Y_1, Y_2, ..., Y_m, and variable x comes from class Y_k. If the size of Y_k is 1, then the original partitioning is valid for Q_{i+1}, and clearly satisfies the properties (P1)-(P3). Otherwise, the size of Y_k is larger than 1 and x is not the representative of Y_k. Let $Y'_k = Y_k \backslash \{x\}$, and let $Y''_k = \{x\}$ be the new classes that are introduced. Clearly, Y''_k satisfies (P1)-(P3). We now consider Y'_k. The representative for this class is the representative for Y_k. It is easy to check that properties (P1)-(P3) hold for Y'_k. **EndProof.**

Theorem 6 *The DSE transformation preserves properties (P1)-(P3).*

Proof: The actual DSE transformation is a sequence of steps, each of which removes only one dead store. Then, this theorem follows immediately from the preservation of partitioning properties in each step shown in Lemma 3. **EndProof.**

In the introductory example, x_2 is the representative variable of its class, hence the dead assignment to x_2 will not be removed by this transformation. Another example is introduced in Fig. 5. The original program P is on the left, and the program Q_0 on the right is the corresponding SSA transformed program whose core set is $\{x_3\}$. There are two non-trivial valid partitions of the three variables of Q_0:

(1) $\{x_1, x_3\}$, $\{x_2\}$: the core variable x_3 is selected as the representative variable of tainted non-core variable x_1. In this case, the dead assignment to x_2 will be removed, since x_2 is from a class of size 1. After assigning a fresh variable name to both x_1 and x_3 by the unSSA transformation, the final program will be as secure as the original program P.

(2) $\{x_1, x_2\}$, $\{x_3\}$: the untainted non-core variable x_2 is selected as the representative variable of tainted non-core variable x_1. In this case, the dead assignment to x_3 will be removed, since x_3 is from a class of size 1. After assigning a

fresh variable name to both x_1 and x_2 by unSSA transformation, no information about password could be leaked and the final program becomes more secure than the original program P.

```
void foo()
{
  int x;

  x = read_password();
  use(x);
  x = 0;
  x = read_password2();
  return;
}
```

```
void foo()
{
  int x;

  x1 = read_password();
  use(x1);
  x2 = 0;
  x3 = read_password2();
  return;
}
```

Fig. 5. `C` program illustrating secure DSE

5 Related Work And Conclusions

The SSA transformation is a standard component of modern compilers, including LLVM and GCC. It continues to be extensively investigated, for its properties and applications to optimization (cf. the "SSA Book" [1]). The fact that it introduces information leaks was noted in our earlier work [6]. In this work, we investigate the question of SSA leaks in some depth, and offer a procedure that blocks the newly introduced leaks by partially reversing the SSA transformation prior to register allocation. Existing work on secure compilation (cf. [8,13]) does not apply to the SSA problem. To the best of our knowledge, the insecurity of SSA and the design of mechanisms that remedy it has not been investigated in the literature. The SWIPE algorithm [9] is a source-to-source transformation which introduces instructions that erase potentially sensitive data after the last use of such data. While the transformation enhances security at the source level, the effect of the new erasure instructions may be negated by the compiler's internal SSA conversion, as illustrated by the introductory example.

The major technical difficulty is to connect the leakage that may occur in a program obtained by a series of optimizations to the leakage introduced by the original conversion to SSA form. Our method tracks these connections using the refinement relations that witness each transformation. Further constraints are needed to ensure that the grouping of variables in the `unSSA` transformation is correct. We give sufficient conditions to show that transformations preserve those constraints, which do hold of the original SSA program, and demonstrate that some common transformations meet these conditions. Loosening the conditions to accommodate more transformations is an important subject for future work.

This work considers a formulation of security that is binary: either a triple is leaky or it is not. This ignores the information content of a leak, making no distinction between, say, the leak of an entire password and the leak of a single character of the password. It is difficult to formulate and analyze the quantitative information content of a leak (cf. [17] for a survey) but one may consider qualitative, knowledge-based formulations (cf. [2]) that are easier to analyze. The construction of a theory that lets one reason about the reduction of the information content of leaks across transformations is an important topic for further research.

This work has not investigated the security of register allocation, which follows the unSSA transform. In LLVM, allocation is done after a straightforward lifting of the program out of SSA form by eliminating phi-functions. Although allocation directly on SSA form is also possible [14], the unSSA transform already produces a non-SSA form program, so those techniques cannot be applied.

Register allocation algorithms, in the main, decide to allocate the live range of a variable either to memory ("spilling") or to a register. Those operations do not introduce a security leak. However, in order to better pack live ranges, an allocator may decide to break up the live range of a variable. This has an effect similar to that of the SSA transform: for instance, the live range of x in the introductory example could be broken up into ranges that correspond to those of the SSA variables x_1 and x_2. These sub-ranges are individually allocated to either registers or memory. In order to preserve security, the allocator should either disable splitting (which could reduce performance) or insert code that clears unused register or memory locations. For example, suppose the range for x is divided up into three sections, corresponding to fresh variables x_1, x_2 and x_3, and x_1 is allocated register A; x_3 is allocated register B; while x_2 is spilled to memory. If x_1 may hold a taint, then the contents of A should be cleared after they are copied to memory. Similarly, when the content in memory is copied to B at the start of x_3, the memory entry should be cleared. Adding those instructions also introduces overhead. However, as splitting is usually attempted around loops [4], it is possible that this overhead is not as considerable as that induced by entirely avoiding splitting. These questions need to be further investigated in an experimental setting.

Acknowledgements. Kedar Namjoshi was supported, in part, by grant CCF-1563393 from the National Science Foundation. We thank our colleagues at Bell Labs and UIC for helpful comments on this work.

References

1. Static single assignment book. http://ssabook.gforge.inria.fr/latest/book.pdf.
2. A. Askarov and S. Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In S. Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 308–322. IEEE Computer Society, 2012.
3. D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations, vol. 1-III. Technical Report ESD-TR-73-278, The MITRE Corporation, 1973.

4. K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In K. Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1383 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 1998.
5. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
6. C. Deng and K. S. Namjoshi. Securing a compiler transformation. In X. Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 170–188. Springer, 2016.
7. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
8. V. D'Silva, M. Payer, and D. X. Song. The correctness-security gap in compiler optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*, pages 73–87. IEEE Computer Society, 2015.
9. K. Gondi, P. Bisht, P. Venkatachari, A. P. Sistla, and V. N. Venkatakrishnan. SWIPE: eager erasure of sensitive data in large scale systems software. In E. Bertino and R. S. Sandhu, editors, *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, pages 295–306. ACM, 2012.
10. S. Hack. Interference Graphs of Programs in SSA Form. Technical Report 2005-15, Universität Karlsruhe, June 2005.
11. V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226. ACM, 2014.
12. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
13. M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. In *CSF*, 2017. (to appear).
14. F. M. Q. Pereira and J. Palsberg. SSA elimination after register allocation. In O. de Moor and M. I. Schwartzbach, editors, *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5501 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2009.
15. A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool (CVT)- automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2(2):192–201, 1998.
16. S. Rideau and X. Leroy. Validating register allocation and spilling. In R. Gupta, editor, *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6011 of *Lecture Notes in Computer Science*, pages 224–243. Springer, 2010.
17. G. Smith. Recent developments in quantitative information flow (invited tutorial). In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 23–31. IEEE, 2015.

18. V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing phi-nodes. In R. K. Cytron and P. Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 62–73. ACM Press, 1995.
19. D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
20. X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.