Open access • Proceedings Article • DOI:10.1145/2854065.2854082

Formal verification of control-flow graph flattening — Source link

Sandrine Blazy, Alix Trieu

Institutions: University of Rennes

Published on: 18 Jan 2016 - Certified Programs and Proofs

Topics: Obfuscation (software), Program transformation, Control flow graph, Formal verification and Compiler

Formal Verification of Control-flow Graph Flattening

Sandrine Blazy, Alix Trieu

Formal Verification of Control-flow Graph Flattening

Sandrine Blazy
IRISA - Université Rennes 1
sandrine.blazy@irisa.fr

Alix Trieu
IRISA - ENS Rennes
alix.trieu@ens-rennes.fr

Abstract

Code obfuscation is emerging as a key asset in security by obscurity. It aims at hiding sensitive information in programs so that they become more difficult to understand and reverse engineer. Since the results on the impossibility of perfect and universal obfuscation, many obfuscation techniques have been proposed in the literature, ranging from simple variable encoding to hiding the control flow of a program.

In this paper, we formally verify in Coq an advanced code obfuscation called control-flow graph flattening, that is used in state-of-the-art program obfuscators. Our control-flow graph flattening is a program transformation operating over C programs, that is integrated into the CompCert formally verified compiler. The semantics preservation proof of our program obfuscator relies on a simulation proof performed on a realistic language, the Clight language of CompCert. The automatic extraction of our program obfuscator into OCaml yields a program with competitive results.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Validation

Keywords program obfuscation, verified compiler, C semantics

1. Introduction

Code obfuscation is emerging as a key asset in security by obscurity. It aims at hiding sensitive information in programs so that they become more difficult to understand and reverse engineer. When an attacker is reverse engineering an obfuscated code, he needs much more effort to extract relevant information from the resulting program [17].

Code obfuscation is used to protect the intellectual property of a software program, or to hide a secret in a piece of software (e.g. keys or watermarks) [9]. For example, code obfuscation is a technique to diversify software, and thus to protect it. Indeed, when a software program is obfuscated in many different ways, each resulting binary code must be attacked separately, dramatically increasing the effort required to reverse engineer these binary codes and to realize that they stem from a same source program [8].

Many obfuscation techniques have been published in the literature (see [8] for a comprehensive survey). Results on the impossibility of perfect and universal obfuscation, such as [3], did not dishearten researchers and practitioners in developing new obfuscation techniques. Well-known examples of basic obfuscations are variable renaming, constant encoding (e.g. multiplying by 3 all integer constants of a program). Other examples of simple obfuscations are variable splitting, array splitting and array merging. Such obfuscations are syntactic and transform data. They can be performed on source code or on compiled code.

Other code obfuscations aim at hiding the control flow of a program. A first example is loop unrolling. Another example is the insertion of opaque predicates. This well-known and efficient example is commonly used to prevent an attacker from relying on static analysis during reverse engineering. An opaque predicate is a conditional expression that always evaluates to the same boolean value, and consists of complex formulae that are difficult to analyze. Inserting an opaque predicate in a statement `s` adds an if-statement to the program such that its condition is the opaque predicate, one of its branches is `s` while the other branch is a sequence of statements that is dead-code, whose purpose is to make the control-flow graph (CFG) more complex.

Advanced code obfuscations are semantic program transformations that hide the control flow. A representative and challenging example that is implemented in state-of-the-art and industrial compilers (e.g. [7, 11, 16]) is CFG flattening. This transformation breaks the CFG of a program by removing all easily identifiable if-statements and loops. These structures are then flattened as they all become different cases of a large switch statement routing the control flow through the statements of the right case (see section 2 for an example program and details).

Usually, the main concern when defining a program obfuscation is the fact that it does not degrade the performance of the compiled code. Surprisingly, very few works in the literature are concerned by semantics preservation of program obfuscations. However, some obfuscations (e.g. CFG flattening) are non-trivial semantic transformations. Even if they are well understood, their implementations on real languages such as C are still error-prone.

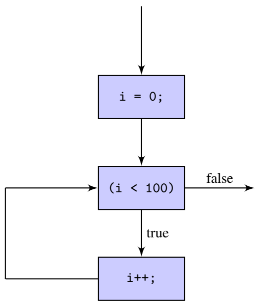
To the best of our knowledge, the only mechanized proof of correctness of some code obfuscations is introduced in [4], where the code obfuscations are basic obfuscations, that operate over a small imperative language. In this paper we go beyond this approach and formally verify an advanced code obfuscation, CFG flattening. Moreover, our semantics-preserving program obfuscator is integrated into the CompCert formally verified compiler [12] and operates over the Clight language of the compiler.

All results presented in this paper have been mechanically verified using the Coq proof assistant [19]. The complete development is available online [1]. This paper makes the three following contributions:

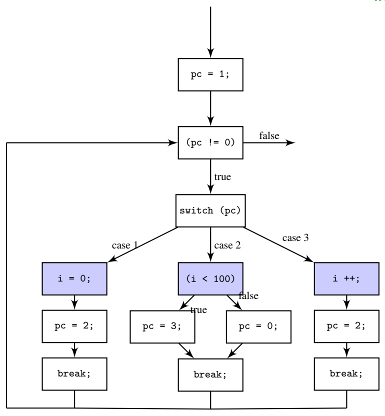
1. a proof of correctness of CFG flattening, an advanced obfuscation formally verified in Coq,
2. a fully executable program obfuscator, integrated into the CompCert C compiler,

```
int i;
i = 0;
while (i < 100) {
  i++;
}
```

(a) Original program



(b) Original CFG



(c) Obfuscated CFG

```
int i;
int pc;
pc = 1;
while (pc != 0) {
  switch (pc) {
    case 1:
      i = 0;
      pc = 2;
      break;
    case 2:
      if (i < 100) {
        pc = 3;
        break;
      } else {
        pc = 0;
        break;
      }
    case 3:
      i++;
      pc = 2;
      break;
  }
}
```

(d) Obfuscated program

Figure 1: Example of CFG flattening: original program with its CFG, the resulting CFG and the resulting program.

3. experiments showing that our obfuscator can be applied on realistic programs.

The remainder of this paper is organized as follows. First, Section 2 briefly introduces our obfuscation based on CFG flattening. Then, Section 3 explains the Clight semantics that is defined in CompCert. The next three sections are devoted to the formal verification of our program obfuscator: Section 4 describes how we formalized CFG flattening, Section 5 explains our main correctness theorem and the simulation relation it relies on, and Section 6 details the main auxiliary lemmata required by the proof. In Section 7, we describe the experimental evaluation of the implementation of our obfuscator. Related work is discussed in Section 8, followed by concluding remarks.

2. Control-flow Graph Flattening

The CFG flattening of a program consists in flattening the control flow of each function by first breaking up the nesting of loops and `if`-statements, and then hiding each of them in a case of a large `switch` statement, that is wrapped inside the body of a loop. Thus, the different basic blocks which were originally at different nesting levels are put next to each other in the CFG. Finally, the control flow of the program is ensured by a variable acting as a program counter that is updated at each execution of a case of the `switch` statement.

Figure 1 shows an example of CFG flattening, where the small program on the left of the figure is obfuscated into the program shown at the right of the figure. The obfuscation removes the `while` loop of the original program. It adds a `switch` statement such that the two assignments and the condition in the original program (i.e. the three basic blocks of the CFG) are hidden in distinct cases of this `switch` statement, which is wrapped inside a loop whose condition depends on the program counter. The CFG is flattened, due to the different cases of the `switch` statement, and the program counter is used to redirect the control flow.

Moreover, a new variable called `pc` is added to route the control flow through the right case of the `switch` statement, and the `switch` statement becomes the body of a loop whose condition depends only on `pc`. Each execution of a case in the `switch` statement consists in executing a statement of the original program, followed by the assignment of `pc` to a new value corresponding to the next case to execute and last a `break` statement to exit the `switch` statement (and hence the loop body). In the end, the execution of the loop stops after executing the last statement of the original program.

The main weaknesses of CFG flattening are twofold: it may degrade the performance of generated code, and the program counter variable may be recognizable during reverse engineering. For these reasons, industrial obfuscators do not perform CFG flattening on whole programs, but only on selected pieces of code. This solution is not specific to control flow obfuscation, and commonly used for all kinds of obfuscations. Most of the time, these pieces of code are selected manually by the programmer who wants to protect for example an algorithm in a larger piece of software.

Other solutions for improving the stealth of CFG flattening is to combine this obfuscation with other simpler obfuscations. Two basic obfuscations are useful for obfuscating the supplementary code added by the CFG flattening: variable splitting that will obfuscate the program counter (e.g. by splitting it into two variables that are updated at different program points) and constant encoding that will change randomly the constant values of the cases in the added `switch` statements, so that they do not correspond to consecutive numbers starting from 1 as in the example of Figure 1.

Our formally verifed CFG flattening transforms C programs. It operates over the Clight language, a simpler version of C where expressions contain no side-effects and there is a single kind of loop. Clight is the first intermediate language of the CompCert compiler. Our CFG flattening is integrated into CompCert as an obfuscation pass. We thus only need to prove the correctness of our obfuscation and then rely on CompCert to get for free the semantics preservation of our obfuscation from source programs to compiled programs.

The proof of correctness of CFG flattening relies on a non-trivial forward simulation theorem stating that each statement of the initial program is also executed in a similar way in the obfuscated program, where the corresponding statement is followed by statements that are added to flatten the control flow of the program. Similarly to other simulation proofs already present in CompCert, the gist of this proof is the definition of the matching relation between program states. Moreover, a peculiarity of our proof (and our matching relation) is that it requires to handle with care the control flow of the initial program (e.g. `continue` and `break` statements in loops), as explained in Section 5.

3. The Clight Semantics in CompCert

This section defines the syntax and semantics of Clight. The proof of correctness of our obfuscator relies on an inductive reasoning on this semantics.

3.1 Syntax

Clight is structured into expressions, statements and functions. Within expressions, all side-effect free operators of C are supported, but not assignment operators nor function calls. Assignments and function calls are presented as statements and cannot occur within expressions. The syntax of Clight is defined in [5]. Since this paper, the Clight language evolved; its syntax is defined in Figure 2. There are now a `statement` and a single kind of loop `(loop s1 s2)` representing infinite loops. It executes `s1` then `s2` repeatedly. A `continue` in statement `s1` branches to statement `s2`. The C loops are derived forms of this infinite loop. For example, the `while` loop that we use in this paper is defined as follows.

```
Definition while(e: expr)(s: statement) :=
 loop ((if(e)then skip else break) ; s) skip.
```

Another recent feature of the Clight language is that an expression can refer to temporary variables, which are a separate class of local variables that do not reside in memory and whose address cannot be taken. As a consequence, there are two kinds of assignments: assignments to a temporary variable and assignments to other variables. In this paper, to simplify the syntax of Clight, we use the same notation `a1 = a2` to denote any of these assignments.

For functions returning "option" types, $\lfloor x \rfloor$ (read: "some x") corresponds to success with return value x, and $\emptyset$ (read: "none") corresponds to failure. In grammars, a^* denotes 0, 1 or several occurrences of syntactic category a, and $a^?$ denotes an optional occurrence of syntactic category a.

Moreover, a `switch` statement consists in an expression and a list of cases. A case is a statement labeled by an integer constant (`case` n) or by the keyword `default` [5]. Given a switch case `sw`, we use the notation `sw[n]` to select (if any) the appropriate case of `sw`, given the value `n` of the selector expression.

Furthermore, in this paper, to simplify the syntax of Clight, we use the same symbol `;` for denoting both a sequence of two statements and also a sequence of switch cases. We also omit in Figure 2 external functions, as we do not transform them, and only show the declaration of internal functions.

3.2 Semantics

The semantics of Clight is defined using a small-step style with continuations, supporting the reasoning on nonterminating programs. It differs from the previous semantics of Clight that was defined in [5] following a big-step style, without continuations. More precisely, the semantics of expressions is defined with a big-step

Expressions:	$a ::=$	id	variable identifier
		$\mid tid$	temporary variable
		$\mid n$	integer constant
		$\mid f$	float constant
		$\mid \texttt{sizeof}(\tau)$	size of a type
		$\mid \texttt{alignof}(\tau)$	alignment of a type
		$\mid op_1\ a$	unary arithmetic op.
		$\mid a_1\ op_2\ a_2$	binary arithmetic op.
		$\mid \texttt{*}a$	pointer dereferencing
		$\mid a.id$	field access
		$\mid \texttt{\&}a$	taking the address of
		$\mid (\tau)a$	type cast
Unary operators:	$op_1 ::=$	$\texttt{-} \mid \texttt{\~{}} \mid \texttt{!} \mid \texttt{absf}$	
Binary operators:	$op_2 ::=$	$\texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}$	arithmetic operators
		$\mid \texttt{<<} \mid \texttt{>>} \mid \texttt{\&} \mid \texttt{\vert} \mid \texttt{\^{}}$	bitwise operators
		$\mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=}$	
		$\mid \texttt{==} \mid \texttt{!=}$	relational operators
Statements:	$s ::=$	$\texttt{skip}$	empty statement
		$\mid a_1 = a_2$	assignment
		$\mid a_1 = a_2(a^*)$	function call
		$\mid a(a^*)$	procedure call
		$\mid s_1; s_2$	sequence
		$\mid \texttt{if}(a)\ s_1\ \texttt{else}\ s_2$	conditional
		$\mid \texttt{switch}(a)\ ls$	multi-way branch
		$\mid \texttt{loop}(s_1, s_2)$	infinite loop
		$\mid \texttt{break}$	exit from current loop
		$\mid \texttt{continue}$	next iteration of the current loop
		$\mid \texttt{goto}\ l$	goto statement
		$\mid \texttt{label}(l, s)$	labeled statement
		$\mid \texttt{return}\ a^?$	return from current function
Switch cases:	$c ::=$	$\texttt{default}$	default case
		$\mid \texttt{case}\ n$	labeled case
List of switch cases:	$ls ::=$	$c : s; ls$	
		$\mid$	empty list
Variables:	$dcl ::=$	$(\tau\ id)^*$	name and type
Functions:	$F ::=$	$\tau\ id(dcl_1)$	dcl_1 = parameters,
		$\{dcl_2;$	dcl_2 = local var.
		$s\}$	body
Programs:	$P ::=$	$dcl;$	global variables
		$F^*;$	functions,
		$\texttt{main} = id$	entry point

Figure 2: Abstract syntax of Clight (excerpt).

style as in [5], and the semantics of statements is a continuation-based small-step semantics, that is defined in Figure 3.

The continuations allow a uniform representation of statement execution and enable us to use the induction principles generated by Coq when reasoning on program executions [2]. They can be seen as a control stack and describe the computations that remain to be performed after the statement under consideration has executed completely. The continuations (of type cont) handle local control flow: Kseq handles sequential execution, Kloop1 (resp. Kloop2) handles the first (resp. second) part of a loop, and Kswitch catches break statements arising out of a switch statement. Kstop represents the termination of the computation. Kcall handles function return; it carries not only a control aspect but an activation record of its own, indicating to the callee function where to store its return value and how to restore the environments of the caller function.

A semantics state (of type state) can either be a call state, a return state or a regular state [13]. Each state has a continuation k representing the statements to execute from this state. A call state C(f,args,k,m) represents the state when one is about to call the function f with arguments args and memory m which basically maps memory addresses with values [15]. Similarly, a return state R(res,k,m) represents a function that returns the value res in memory state m. Regular states S(f,s,k,e,le,m) are used during the execution of the current function f. There are two kinds of environment, e maps variables with their memory addresses where their values are stored at, le maps variables directly with their values but only for variables whose addresses are never taken. This is useful for optimizations during compilation as these variables do not need to reside in the memory and can be put in registers.

The Clight small-step semantics for statements is defined as an inductive predicate called step. It consists of 17 semantic rules for evaluating expressions and 25 rules defining the execution of statements, together with many other rules devoted to C unary and binary operators and also memory stores and loads. In this paper, for clarity reasons, we consider step as a transition relation between two states and omit the trace recording the input/output events that are triggered during transitions.

As an example, we only show a few transition rules defined in the step semantics. The step_set rule states that the statement id=a assigning a temporary variable id modifies the value of variable id to v (if a evaluates to v in the environment provided by e, le and m) and updates the local environment le accordingly. The step_seq rule states that to execute the statement s1;s2, the statement s1 needs to be executed first and the statement s2 is added to the continuation k to form Kseq s2 k and execute s2 later. When s1 finishes its execution, the rule step_skip_seq is used to remove s2 from the continuation and promote it to the next statement s to be executed.

The next rules of Figure 3 define the execution of an infinite loop (loop s1 s2). The first part s1 of the loop is first executed (rule step_loop), and the continuation k becomes (Kloop1 s1 s2 k) when entering the loop. When all the statements of s1 are executed, s1 is reduced into the skip statement, and the rule step_s_or_c_loop1 is triggered to execute the second part s2 of the loop. Again, the continuation starting with Kloop2 indicates the entry in this second part. When all the statements of s2 are executed, s2 is reduced into the skip statement, and the rule step_skip_loop2 is triggered to execute again the statements of the loop. The only way to exit the loop is to execute a break statement. For example, the rule step_break_loop1 states that the execution of a break statement during the execution of s1 reduces the break into skip and triggers the exit from the loop (i.e. s1 and s2 are skipped and the continuation (Kloop1 s1 s2 k) becomes k).

```
Inductive cont: Type :=
| Kstop                          (* Program ended *)
| Kseq(s2:statement) (k:cont)
                               (* after s1 in s1;s2 *)
| Kloop1(s1:statement) (s2:statement) (k:cont)
                          (* after s1 in loop s1 s2 *)
| Kloop2(s1:statement) (s2:statement) (k:cont)
                    (* after s2 in loop s1 s2 *)
| Kswitch (k:cont)   (* catches break statements *)
| Kcall(oi:option ident)(* where to store result *)
      (f: function) (* calling function *)
      (e:env)       (* local env of f *)
      (le:temp_env) (* temporary env of f *)
      (k:cont).
Inductive state: Type :=
 | C(f: function)(args: list val)(k: cont)(m: mem)
 | R(res: val)(k: cont)(m: mem)
 | S(f: function)(s: statement)(k: cont)(e: env)
            (le: temp_env)(m: mem).
Inductive step: state -> state -> Prop :=
| step_set: forall f id a k e le m v,
    eval_expr e le m a v ->
    step (S f (id=a) k e le m)
         (S f skip k e (PTree.set id v le) m)
| step_seq: forall f s1 s2 k e le m,
    step (S f (s1;s2) k e le m)
         (S f s1 (Kseq s2 k) e le m)
| step_skip_seq: forall f s k e le m,
    step (S f skip (Kseq s k) e le m)
         (S f s k e le m)
| step_break_seq: forall f s k e le m,
    step (S f break (Kseq s k) e le m)
         (S f break k e le m)
| step_continue_seq: forall f s k e le m,
    step (S f continue (Kseq s k) e le m)
         (S f continue k e le m)
| step_loop: forall f s1 s2 k e le m,
    step (S f (loop s1 s2) k e le m)
         (S f s1 (Kloop1 s1 s2 k) e le m)
| step_s_or_c_loop1: forall f s1 s2 k e le m x,
    x = skip \/ x = continue ->
    step (S f x (Kloop1 s1 s2 k) e le m)
      (S f s2 (Kloop2 s1 s2 k) e le m)
| step_break_loop1: forall f s1 s2 k e le m,
    step (S f break (Kloop1 s1 s2 k) e le m)
      (S f skip k e le m)
| step_skip_loop2: forall f s1 s2 k e le m,
    step (State f skip (Kloop2 s1 s2 k) e le m)
      (S f (loop s1 s2) k e le m)
| step_break_loop2: forall f s1 s2 k e le m,
    step (S f break (Kloop2 s1 s2 k) e le m)
      (S f skip k e le m)
| step_switch: forall f a ls k e le m v n,
    eval_expr e le m a v ->
    sem_switch_arg v (typeof a) = Some n ->
    step (State f (switch a ls) k e le m)
      (State f (select_switch n ls)
             (Kswitch k) e le m)
| step_skip_break_switch: forall f x k e le m,
    x = skip \/ x = break ->
    step (State f x (Kswitch k) e le m)
     E0 (State f skip k e le m)
```

Figure 3: Small-step semantics of Clight (excerpt).

The last two rules define the execution of a (switch a ls) statement. The expression a is first evalued to some value v, which is then interpreted either as a 32-bit or 64-bit unsigned integer according to the type of a. The function select_switch is then used to find the corresponding case in ls and transforms it into sequences of statements to execute next. As for previous statements, once this sequence of statements is executed, it is reduced into the skip statement, and the rule step_skip_break_switch is triggered.

In the rest of this paper, the reflexive transitive closure of the step relation is denoted by star, and its transitive closure is denoted by plus. The set of reachable states of a program P is denoted by reach(P) = star s0 s', where s0 represents an initial state. Let us note that diverging programs are modeled as infinite sequences of reduction steps.

4. Formalization of Control-Flow Graph Flattening

CFG flattening is a program transformation that modifies the control flow of the program. Indeed, the body of the program is broken up into multiple basic blocks and then encapsulated into a switch statement with each block constituting a different case. The switch statement has then the responsibility of selecting the right case to execute, which is ensured by an additional variable that acts as a program counter. Finally, the switch statement is then encapsulated into a loop statement in order to simulate the program (e.g. see Figure 1).

Figure 4 shows an excerpt of our obfuscation that performs CFG flattening on a program and generates an obfuscated program. Obfuscating a program consists in obfuscating each of its functions, which requires to find in each function a fresh local variable pc, and then to call (obf_body pc body(f)) to add the declaration of pc and to obfuscate all the statements of the function body. The variable pc is declared as an unsigned int, which corresponds to a 32-bit unsigned machine integer in CompCert. So, if the statements to obfuscate produce more than 2^{32} cases (which never happens in practice), an error is raised as some cases would not be able to be reached as their corresponding number would be too high. Thus, to report an error, the transformation functions make use of an error monad res A, as shown for obf_body for example. A value in res A is either an Error msg where msg is a string, or an OK a with a being a value of type A.

As in the example in Figure 1, the obfuscated program is a loop whose condition is (pc!=0) and whose body is a switch statement switching on the values of pc and such that the different case statements correspond to the flattened statements. More precisely, each basic statement s (i.e. an assignment, a function call, a return statement or a skip statement) is flattened into a case statement (case counter: s; pc=k; break). The flattening of statements is performed by the function called flatten that is working out how to correctly number the switch cases. This function also makes use of the error monad and must thus be written in a monadic style. However, for the sake of simplicity, we do not explicitly write the binding operations that are needed when using recursive calls, and omit cases that yield Error msg.

Moreover, break and continue do not have a defined semantics outside of loops, thus flatten will raise an error if it encounters such a statement. The current version of our obfuscator does not handle programs with switch statements.

The flattening of other statements calls recursively the flattening of all the basic statements. For instance, the flattening of a sequence s1;s2 consists in generating 1) the switch cases corresponding to the flattening of each basic statement of s1 and 2) the switch cases corresponding to the flattening of each basic statement of s2

```
Definition obf (P: program) := ...
(* For each function f of P, call obf_fun f *)

Definition obf_fun (f:function) := ...
(* find in f a fresh local variable pc
   call obf_body pc body(f)          *)

Definition obf_body (pc: ident) (body: statement) :=
  if |body| >= 2^32 then
    Error ("Program too big")
  else
    OK (unsigned int pc = 1;
        while (pc != 0) {
          switch (pc) (flatten pc body 1 0)
        })

Fixpoint flatten (pc: ident)(s: statement)
                 (counter k: Z) :=
match s with
| skip => (case counter: skip; pc=k; break)
| a = b => (case counter: a=b; pc=k; break)
| s1; s2 =>
    (flatten pc s1 (counter) (counter+|s1|));
    (flatten pc s2 (counter+|s1|) k)
| if b then s1 else s2 =>
  (case counter: if b then pc=counter+1
                 else pc=counter+1+|s1|; break);
  (flatten pc s1 (counter+1) k);
  (flatten pc s2 (counter+1+|s1|) k)
| loop s1 s2 =>
  (case counter: pc=counter+1; break);
  (flatten_loop pc s1 (counter+1)
        (counter+1+|s1|) (counter+1+|s1|) k);
 (flatten_loop pc s2 (counter+1+|s1|) counter 0 k)
| break | continue => Error
| ...
end.

Fixpoint flatten_loop (pc: ident)(s: statement)
            (counter k to_continue to_break: Z) :=
  match s with
  | continue =>
      (case counter: pc = to_continue; break)
  | break => (case counter: pc = to_break; break)
  ...
  end.
```

Figure 4: Code of our CFG flattening obfuscation (excerpt).

(i.e. starting from a counter equal to counter+|s1|), where |s| denotes the length of the statement s (i.e. the number of its basic statements).

In the expression (flatten pc body counter k), the parameter called counter represents the initial value of the program counter (i.e. it is the number of the first switch case of the generated sequence of switch cases). The k parameter behaves as a continuation and represents the value of the next statement to execute (according to the control flow of the original program). This parameter is required to handle the recursive calls of the flatten function.

In order to flatten an infinite loop (loop s1 s2), the expression (flatten pc (loop s1 s2) counter k) first generates a switch case numbered by counter and modifies pc to counter+1. Then, it flattens s1 by numbering the flattened statements from `counter+1` to `(counter+|s1|)` and its last case is asked to modify the `pc` variable to `(counter+|s1|+1)`, which corresponds to the first number of the flattening of `s2`. Last, `s2` is flattened such that at its end the `pc` variable is modified back to `counter` to correctly model the control flow of the loop. The precise form of the transformation of infinite loops is important and will be referred back to in the next section.

Furthermore, the function `flatten` uses an auxiliary function called `flatten_loop` which is the same as `flatten` except that it now knows how to transform `break` and `continue` statements thanks to the `to_break` and `to_continue` extra arguments, which indicate how to renumber the `pc` variable.

The only way to get out of a loop is through a `break` statement. Thus, in a loop, a `break` statement is transformed into a `pc=k` statement, with `k` representing where the control flow is supposed to go after the loop statement has finished executing. Similarly, a `continue` in `s1` is transformed into a `pc=counter+1+|s1|` statement. However, `continue` statements are guaranteed not to appear in `s2` as they do not have a semantics defined there, thus whatever the argument given as `to_continue` has no impact.

In the definition of `obf_body` in Figure 4, the expression `(flatten pc body 1 0)` produces the sequence of switch cases corresponding to the program `body` by assigning the number 1 to its first statement and telling that the last case must change the value of `pc` to 0, which corresponds to the exit condition of the loop that encapsulates the `switch` statement. For example, the expression `(flatten pc skip 1 0)` produces the following switch case: `(case 1: skip; pc=0; break)`.

5. Correctness of Control-Flow Graph Flattening

This section defines our main correctness theorem and explains the main difficulties encountered during the corresponding proof. First, we describe our main simulation theorem and explain our solution to apply it on all kinds of program executions that are defined in the Clight semantics: terminating executions, but also infinite executions (called diverging executions in CompCert). Then, we detail the different matching relations we have defined in order to prove the simulation theorem.

5.1 Main Simulation Theorem

To prove that the CFG flattening preserves the Clight semantics, we rely on a standard technique used throughout CompCert and show a forward simulation diagram expressed in the following theorem. Given an initial program `P1` and its corresponding obfuscated program `P2`, it states that each transition step in `P1` must correspond to transitions in `P2` and preserve as an invariant a relation between states of `P1` and `P2`.

Theorem 1. *Let `P1` be a program and `P2` its corresponding obfuscated program (i.e. `P2=obf(P1)`). Then, we have the following simulation relation between reachable states of both programs (i.e. `s1,s1'` ∈ `reach(P1)` and `s2` ∈ `reach(P2)`): for each step `(step s1 s1')` of the execution of `P1`, and each state `s2` of the execution of `P2` that matches with `s1`, there exists a state `s2'` that matches with `s1'` and that is reached after zero, one or several steps from `s2`.*

In Coq, we call `match_states` the matching relation between states. The simulation theorem is written in Coq in Figure 5. The theorem uses a measure in order to handle diverging execution steps. The measure is defined over transition states. It is however only used for regular states, which contain information on the statement to execute, its continuation and environments. It is difficult to guess how environments are modified during the execution of the program, the measure thus actually only uses the statement and its continuation to define itself.

```
Theorem simulation: forall (s1 s1':state),
 step s1 s1' ->
 forall (s2:state), match_states s1 s2 ->
 (exists s2', plus s2 s2' /\ match_states s1' s2')
 \/(measure s1'<measure s1 /\ match_states s1' s2).
```

$$\begin{tikzcd} s_1 \arrow[r, dash, "\sim"] \arrow[d] & s_2 \arrow[d, dashed, "+"'] \\ s_1' \arrow[r, dash, dashed, "\sim"'] & s_2' \end{tikzcd} \qquad \begin{tikzcd} s_1 \arrow[r, dash, "\sim"] \arrow[d] & s_2 \arrow[dl, dash, dashed, "\sim"] \\ s_1' & \end{tikzcd}$$

with $m(s_1') < m(s_1)$

Figure 5: Simulation relation (in Coq) and corresponding simulation diagram. Plain lines are hypotheses, dashed lines are conclusions.

More precisely, there are two situations in the simulation diagrams that we prove (hence the 'or' at the beginning of the last line in the Coq theorem); they are depicted in Figure 5. First (left of Figure 5), as explained in the previous section, most statements of `P1` are flattened into a non-empty sequence of statements (e.g. an assignment is flattened into a sequence of three statements that is wrapped inside a switch case). So, the simulation diagram states that for two matching states `s1` and `s2`, the execution `(step s1 s1')` of such a statement in `P1` must match with the execution `(plus s2 s2')` of the corresponding sequence of statements, and the resulting states `s1'` and `s2'` must match as well.

Second (right of Figure 5), some other statements of `P1` are not transformed and do no exist in `P2`. For such statements, the simulation diagram states that an execution `(step s1 s1')` in `P1` corresponds to no execution in `P2`, that is we must have `(star s2 s2')`, as the star relation comprises empty executions. In such situations, we use a measure associated with the execution states of `P1` that decreases to ensure that there are no infinite stuttering steps (that can happen because of diverging executions, see [13]). Indeed, if `P1` diverges (i.e. there are infinitely many transition steps), then `P2` must also diverge. The measure m is a natural integer and therefore cannot decrease indefinitely, which thus ensures that infinitely many transition steps are also simulated by infinitely many transition steps.

There are a few cases where a transition in the original program corresponds to no transition in the transformed program, the main one is when a statement is reduced to `skip` in the original program, this `skip` is purely semantic and does not appear syntactically in the program, it thus has no corresponding case in the transformed program. The second case is `step_break_seq` (see Figure 3), the original program pops the continuation until it finds a `Kloop` or `Kswitch` to catch the `break` statement, whereas the transformed program already knows which case to execute. Similarly with `step_continue_seq`, there is no corresponding step in the transformed program. The last case is `step_seq`, the original must unfold `s1;s2` to know that the next statement to execute is `s1` whereas the transformed program sees no such thing as it has been flattened and already knows that `s1` is the next statement to execute. Consequently, the measure `m` that we define must respect the following conditions.

```
m(s, k) < m(skip, Kseq s k)
m(s1, Kseq s2 k) < m(s1;s2, k)
m(continue, k) < m(continue, Kseq s k)
m(break, k) < m(break, Kseq s k)
```

```
Fixpoint num_stmt (s: statement): nat :=
  match s with
    | s1;s2 => num_stmt s1 + num_stmt s2 + 2
    | _ => 0
  end.

Fixpoint num_cont (k: cont): nat :=
  match k with
    | Kseq s k => num_cont k + num_stmt s + 1
    | Kstop | Kcall _ _ _ _ _ | Kswitch _ => 0
    | Kloop1 _ s2 k => num_cont k + num_stmt s2 + 2
    | Kloop2 _ _ k => num_cont k + 1
  end.

Definition measure (st: state): nat :=
  match st with
    | State _ s k _ _ _ => num_stmt s + num_cont k
    | _ => 0
  end.
```

Figure 6: Measure defined to avoid infinite stuttering steps in the simulation proofs.

```
m(s', Kloop2 s s' k) < m(skip, Kloop1 s s' k)
m(loop s s', k) < m(skip, Kloop2 s s' k)
```

One may wonder why the `flatten` function adds an extra case (`case counter:pc=counter+1;break`) which only modifies the program counter variable when loops are flattened. This is actually used to simulate the transition step `step_loop` in Figure 3. It would otherwise have no corresponding transition step without the extra case that we add, and would add an additional condition that `m` must respect `m(s1,Kloop1 s1 s2 k) < m(loop s1 s2,k)`. When combined with the previous ones, one can derive an absurdity.

```
m(s1,Kloop1 s1 s2 k) < m(loop s1 s2,k)
m(s2,Kloop2 s1 s2 k) < m(skip,Kloop1 s1 s2 k)
m(loop s1 s2,k) < m(skip,Kloop2 s1 s2 k)
```

Indeed, if we consider `s1` and `s2` to be both `skip` statements, then we can derive the following inequality which is absurd: `m(loop skip skip,k) < m(loop skip skip,k)`. Our solution was thus to introduce the extra cases during CFG flattening, in order to be able to define a measure. It is called `measure` and detailed in Figure 6.

5.2 Matching Relations Used in the Simulation Theorem

The gist of the proof is to define the matching relation $\sim$ (called `match_states` in Coq) matching a state s in `P1` with a state s' in `P2` such that s' is similar to s except that the value of the program counter corresponds to the correct statement to execute. The difficulty is to properly state what it means for the program counter to correspond to the correct statement to execute.

Informally, one can think of the `pc=n;break` statements in the switch cases as arrows in a CFG linking basic blocks to others, as this is roughly what the `switch` statement wrapped inside the `while` statement allows us to do. Figure 7 illustrates an example of this matching, where each node in the initial CFG must match to the corresponding switch case in the obfuscated program.

Indeed, the first switch case is linked to the second one, which tests whether the condition `(i<100)` holds and goes to the third switch case if it is true, or modifies the program counter to zero otherwise (which corresponds to the exit condition of the loop). It is exactly the same information that is exhibited by the CFG. This

```
case 1:
  i = 0;
  pc = 2;
  break;
case 2:
  if (i < 100) {
    pc = 3;
    break;
  } else {
    pc = 0;
    break;
  }
case 3:
  i = i + 1;
  pc = 2;
  break;
```

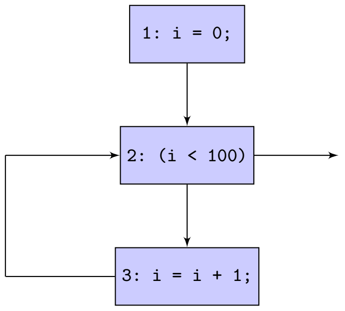


Figure 7: Matching between switch cases and initial CFG.

way, the proof can be seen as explaining how to rebuild the CFG from the switch cases that were generated by the transformation. It remains relatively simple on a small example, but our simulation proof must actually relate the abstract syntax tree of the program with its flattened version and not the CFG with its flattened version.

The $\sigma \sim \sigma'$ relation between two states σ and σ' is defined in Figure 8 by three rules corresponding to the matching between either call states or return states or regular states. Two call states can be matched when they share the same list of arguments (as our obfuscation does not change the arguments in functions) and the same memory. Indeed, the program counter that we add in the obfuscated program does not reside in the memory as its address is never taken, so the code added by the obfuscation does not change the memory. To match a call state `C(f,a,k,m)` with a call state `C(f',a,k',m)`, we need to check that `f'` is the transformed version of function `f` and that the call continuations inside `k` and `k'` can be matched (i.e. `k` $\simeq$ `k'`). This corresponds to rule (1) in Figure 8.

This matching $\simeq$ between call continuations is not detailed in this paper. Informally, a call continuation is either `Kstop` or a `Kcall`. A `Kstop` must be matched with a `Kstop`, while a `(Kcall oid f e le k)` must be matched with a `(Kcall oid f' e le' k')` such that `f'` is the transformed version of `f` and the temporary environment `le'` is `le` except that the program counter is defined in `f'` and has a value. Furthermore, as the continuation `k` contains the next statements to execute, `k'` is the corresponding transformed continuation. Return states are similarly matched (see rule (2) in Figure 8).

The last case matches a regular state `S(f,s,k,e,le,m)` of the initial program with a regular state `S(f',s',k',e,le',m)` of the obfuscated program. Again, `f'` is the transformed version of `f`, the continuations `k` and `k'` must match (i.e. `k` $\simeq$ `k'`) (see rule (3)). Once again, the memory and the local environment do not need to be modified. Let us note that the continuation `k'` must be either `Kstop` or a `Kcall`. Indeed, the whole point of the CFG flattening transformation is to make the control flow of the program harder to understand, thus in the transformed program, the continuations strictly become a call stack. As a consequence, the only possible continuations in a regular state of the transformed program are `Kstop` and `Kcall` (i.e. they can not be `Kseq` or `Kloop`).

Moreover, the local environment `le'` is `le` except that the program counter pc_f has a value and it is n, the number of the correct switch case (i.e. $le' = le \dagger \{\texttt{pc}_f \mapsto \lfloor n \rfloor\}$). The correct switch case is the one that corresponds to `s` in `s'`, the while loop wrapping the

$$\frac{\mathtt{obf}(\mathtt{f}) = \lfloor f' \rfloor \qquad k, k' \in \{\mathtt{Kstop}, \mathtt{Kcall}\} \qquad k \simeq k'}{C(f, a, k, m) \sim C(f', a, k', m)} \quad (1) \qquad\qquad \frac{\mathtt{obf}(\mathtt{f}) = \lfloor f' \rfloor \qquad k, k' \in \{\mathtt{Kstop}, \mathtt{Kcall}\} \qquad k \simeq k'}{R(v, k, m) \sim R(v, k', m)} \quad (2)$$

$$\frac{\begin{array}{c} \mathtt{obf}(\mathtt{f}) = \lfloor f' \rfloor \qquad k' \in \{\mathtt{Kstop}, \mathtt{Kcall}\} \\ k \simeq k' \qquad le' = le \dagger \{\mathtt{pc}_f \mapsto \lfloor n \rfloor\} \qquad \mathtt{flatten}(\mathtt{pc_f}, \mathtt{body}(\mathtt{f}), 1, 0) = \lfloor ls \rfloor \qquad s' = \mathtt{while}(\mathtt{pc}_f \,!= 0)(\mathtt{switch}\,\mathtt{pc}_f\,\mathtt{ls}) \\ \forall s_1, s_2, k'', \mathtt{context}(k) \in \{\mathtt{Kloop1}\, s_1 s_2 k'', \mathtt{Kloop2}\, s_1 s_2 k''\} \implies \exists n_0 \text{ such that} \mathtt{pc}_f, k'' \vdash (\mathtt{loop}\, s_1 s_2) \sim ls[n_0] \wedge n_0, \mathtt{pc}_f, k \vdash s \approx ls[n] \\ \mathtt{context}(k) \in \{\mathtt{Kstop}, \mathtt{Kcall}\} \implies \mathtt{pc}_f, k \vdash s \sim ls[n] \end{array}}{S(f, s, k, e, le, m) \sim S(f', s', k', e, le', m)} \quad (3)$$

Figure 8: Matching between states ($\sigma \sim \sigma'$ relation).

obfuscated sequence of statements corresponding to `s`, that always has the following shape.

```
while (pc != 0) {
    switch (pc) {
        case 1: ...
        case 2: ...
        ...
        case m: ...
    }
}
```

In this `while` loop, the sequence *ls* of switch cases is the body of `f` broken into small blocks. The difficulty of the proof lies in how to relate the control flow in the original program with the syntactic elements of the transformed program.

To that purpose, we define another matching relation written $\mathtt{pc}, k \vdash s \sim ls[n]$ between a statement s of the original program and its corresponding sequence of statements $ls[n]$ in the transformed program (i.e. it is the sequence of the case numbered n in the added switch statement), given a continuation k and the program counter `pc` that the transformation adds in the obfuscated program. This relation is defined by the rules (4) to (9) of Figure 9.

The first rule (numbered (4)) defines the matching between an assignment and its obfuscated switch case. Besides the definition of this switch case, the rule requires the following hypothesis: $\mathtt{pc}, k \vdash \mathtt{skip} \sim ls[\mathtt{next\_stmt}]$. It stems from the transition step called `step_set` in the semantics (see Figure 3) that reduces an assignment into a skip statement, whatever the continuation. Thus, the continuation is only checked in a next step, when `skip` will be executed. For example, the `step_skip_seq` rule in Figure 3 indicates that after a `skip`, if the continuation is a `Kseq s k`, the next instruction to execute is `s`.

However, this is problematic for our proof. As previously stated, one cannot distinguish a `skip` that is syntactically present in the code from one that is purely semantic. Thus, we must be able to handle both situations, which corresponds respectively to the rules (5) and (6). If the `skip` is present in the original program, then we know that there must be a corresponding case in the transformed program which corresponds to rule (5). On the other hand, if the `skip` is semantic and does not appear in the program, then rule (6) applies and we must prove that the program counter already points to the next statement to execute.

The matching relation for sequences of statements is defined in rule (7). Similarly to rule (6), there is no corresponding switch case to the transition `step_seq`, thus we have to also prove that the program counter already points to the correct switch case corresponding to `s1`.

Loops modify the control flow of the program and are more difficult to handle. Executing a loop `(loop s1 s2)` consists in first executing `s1` (rule `step_loop` in Figure 3), then executing `s2` (rule `step_s_or_c_loop1` in Figure 3). Afterwards, the loop

$$\frac{\begin{array}{c} ls[n] = (e_1 = e_2; pc = \mathtt{next\_stmt}; \mathtt{break}) \\ \mathtt{pc}, k \vdash \mathtt{skip} \sim ls[\mathtt{next\_stmt}] \end{array}}{pc, k \vdash e_1 = e_2 \sim ls[n]} \quad (4)$$

$$\frac{\begin{array}{c} ls[n] = (\mathtt{skip}; pc = \mathtt{next\_stmt}; \mathtt{break}) \\ pc, k \vdash s \sim ls[\mathtt{next\_stmt}] \end{array}}{pc, \mathtt{Kseq}\, s\, k \vdash \mathtt{skip} \sim ls[n]} \quad (5)$$

$$\frac{pc, k \vdash \mathtt{s} \sim ls[n]}{pc, \mathtt{Kseq}\, s\, k \vdash \mathtt{skip} \sim ls[n]} \quad (6)$$

$$\frac{pc, \mathtt{Kseq}\, s_2\, k \vdash s_1 \sim ls[n]}{pc, k \vdash \mathtt{s_1}; \mathtt{s_2} \sim ls[n]} \quad (7)$$

$$\frac{\begin{array}{c} ls[n] = (\mathtt{if}\ b\ \mathtt{then}\ pc = n+1\ \mathtt{else}\ pc = n+1+|\mathtt{s_1}|; \mathtt{break}) \\ pc, k \vdash \mathtt{s_1} \sim ls[n+1] \qquad pc, k \vdash \mathtt{s_2} \sim ls[n+1+|\mathtt{s_1}|] \end{array}}{pc, k \vdash \mathtt{if}\ b\ \mathtt{then}\ \mathtt{s_1}\ \mathtt{else}\ \mathtt{s_2} \sim ls[n]} \quad (8)$$

$$\frac{\begin{array}{c} ls[n] = (pc = n+1; \mathtt{break}) \\ n, pc, \mathtt{Kloop1}\, s_1\, s_2\, k \vdash s_1 \approx ls[n+1] \end{array}}{pc, k \vdash \mathtt{loop}\, \mathtt{s_1}\, \mathtt{s_2} \sim ls[n]} \quad (9)$$

Figure 9: Matching between statements ($\mathtt{pc}, k \vdash s \sim ls[n]$ relation).

is executed again (rule `step_skip_loop2` in Figure 3). It is however possible to exit a loop by using a `break` statement (rules `step_break_loop1` and `step_break_loop2` in Figure 3). Thus, similarly to the previous rules defined for sequences, one would probably want to define the matchings numbered (w1), (w2), and (w3) in Figure 10.

But, this does not work because of circular reasoning. Indeed, the matching $\mathtt{pc}, k \vdash \mathtt{loop}\, s_1\, s_2 \sim ls[n]$ requires the matching $\mathtt{pc}, (\mathtt{Kloop1}\, s_1\, s_2\, k) \vdash s_1 \sim ls[n+1]$, which in turn requires the matching $\mathtt{pc}, (\mathtt{Kloop2}\, s_1\, s_2\, k) \vdash s_2 \sim ls[n+1+|s1|]$, which requires again the initial matching $\mathtt{pc}, k \vdash \mathtt{loop}\, s_1\, s_2 \sim ls[n]$. Our solution is to modify rule (w1) into rule (9), (w2) into rule (12) in Figure 11 and (w3) into rule (13).

However, it is not enough as we are now lacking information to match `S(f,(loop s1 s2),k,e,le,m)` after the transition step from `S(skip,(Kloop2 s1 s2 k),e,le,m)` to the state `S(f,(loop s1 s2),k,e,le,m)`, as `(loop s1 s2)` is not matched with anything anymore. The solution is that when trying to match a regular state `S(f,s,k,e,le,m)`, if the continuation `k` is either a `(Kloop1 s1 s2 k')` or a `(Kloop2 s1 s2 k')`, then we need to remember that the corresponding `(loop s1 s2)` can be matched.

$$\frac{ls[n] = (pc = n+1; \texttt{break}) \quad pc, \texttt{Kloop1}\, s_1\, s_2\, k \vdash s_1 \sim ls[n+1]}{pc, k \vdash \texttt{loop}\, \texttt{s}_1\, \texttt{s}_2 \sim ls[n]} \quad \text{(w1)}$$

$$\frac{ls[n] = (\texttt{skip}; pc = \texttt{next\_stmt}; \texttt{break}) \quad pc, k \vdash \texttt{loop}\, \texttt{s}_1\, \texttt{s}_2 \sim ls[\texttt{next\_stmt}]}{pc, \texttt{Kloop2}\, s_1\, s_2\, k \vdash \texttt{skip} \sim ls[n]} \quad \text{(w2)}$$

$$\frac{pc, k \vdash \texttt{loop}\, \texttt{s}_1\, \texttt{s}_2 \sim ls[n]}{pc, \texttt{Kloop2}\, s_1\, s_2\, k \vdash \texttt{skip} \sim ls[n]} \quad \text{(w3)}$$

Figure 10: Wrong matching relations.

So, we define in rules (10) to (13) a new matching relation $n_0, \texttt{pc}, k \vdash s \approx ls[n]$ for loops, where we add the number n_0 of the case corresponding to the loop entry. The whole matching relation $\approx$ consists of rules (10) to (13) plus rules (4') to (9') that are similar to rules (4) to (9), but with an added n_0. Such an n_0 is exhibited in a precondition of rule (3). Indeed, in this rule, the continuation k contains the context in which the current statement is executed, i.e. whether it is within a loop or not. This precondition thus states that if the context is a loop, then we must know a n_0 such that $\texttt{pc}, k'' \vdash \texttt{loop} s_1 s_2 \sim ls[n_0]$. In other words, we must remember which number was assigned to the start of the loop, which replaces the preconditions that were present in (w2) and (w3), hence removing the circular reasoning problem.

Rule (14) gives the matching relation for a break statement present in the second part of a loop. The switch case corresponding to such a break only modifies the program counter, and the value it is modified to must point to the next statement after the loop. This is indicated by the $\texttt{pc}, k \vdash \texttt{skip} \sim ls[\texttt{next\_stmt}]$ requirement. Note that the relation used is $\sim$ as we are now out of the loop. But this is only the case if there are no nested loops which we assumed in this paper for the sake of simplicity. In the general case, instead of using an integer n_0, we use a stack of such integers. Each time a loop is entered, a new number is pushed on the stack. Each time a break is encountered, the corresponding value is popped from the stack.

6. Auxiliary Lemmata

CFG flattening is a peculiar program transformation in that it also creates a new variable which serves as a program counter that is used to dispatch the control flow. Most program transformations that were already present in CompCert – which is primarily a compiler – do not need to introduce new variables. Thus, it was necessary to define how to create fresh variables and prove that they are actually fresh. Variables in CompCert are internally represented by identifiers which are actually positives, the Coq type for strictly positive integers.

Hence, to define a new variable for a function, one must find out the greatest positive that appears in the body of the function. This can be accomplished by scanning through the abstract syntax tree of the function. The successor of this identifier is thus guaranteed to never appear in the body of the function as it is strictly greater than every positive that may appear. We thus have the two following lemmata.

```
Lemma max_ident_not_in_stmt: forall s id,
    (max_ident_in_stmt s 1) < id ->
    not_in_stmt id s.
```

$$\frac{ls[n] = (e_1 = e_2; pc = \texttt{next\_stmt}; \texttt{break}) \quad n_0, \texttt{pc}, k \vdash \texttt{skip} \approx ls[\texttt{next\_stmt}]}{n_0, pc, k \vdash e_1 = e_2 \approx ls[n]} \quad \text{(4')}$$

$$\dots \quad \text{(9')}$$

$$\frac{ls[n] = (\texttt{skip}; pc = \texttt{next\_stmt}; \texttt{break}) \quad n_0, pc, \texttt{Kloop2}\, s_1\, s_2\, k \vdash s_2 \approx ls[\texttt{next\_stmt}]}{n_0, pc, \texttt{Kloop1}\, s_1\, s_2\, k \vdash \texttt{skip} \approx ls[n]} \quad (10)$$

$$\frac{n_0, pc, \texttt{Kloop2}\, s_1\, s_2\, k \vdash s_2 \approx ls[n]}{n_0, pc, \texttt{Kloop1}\, s_1\, s_2\, k \vdash \texttt{skip} \approx ls[n]} \quad (11)$$

$$\frac{ls[n] = (\texttt{skip}; pc = n_0; \texttt{break})}{n_0, pc, \texttt{Kloop2}\, s_1\, s_2\, k \vdash \texttt{skip} \approx ls[n]} \quad (12)$$

$$\frac{}{n_0, pc, \texttt{Kloop2}\, s_1\, s_2\, k \vdash \texttt{skip} \approx ls[n_0]} \quad (13)$$

$$\frac{ls[n] = (pc = \texttt{next\_stmt}; \texttt{break}) \quad pc, k \vdash \texttt{skip} \sim ls[\texttt{next\_stmt}]}{n_0, pc, \texttt{Kloop2}\, s_1\, s_2\, k \vdash \texttt{break} \approx ls[n]} \quad (14)$$

Figure 11: Matching between statements ($n_0, \texttt{pc}, k \vdash s \approx ls[n]$ relation).

```
Lemma new_ident_not_in_body:  forall f,
 not_in_stmt (new_ident_for_function f)(fn_body f).
```

The expression (max_ident_in_stmt s 1) returns the greatest identifier it can find in the statement s and defaults to 1 if there is none (typically if s is skip). Thus, the first lemma states that any identifier greater than (max_ident_in_stmt s 1) is guaranteed to not appear in s.

The second lemma states that (new_ident_for_function f) does not appear in the body of the function f and is a direct consequence of the first lemma. This proof is fairly easy, but is crucial to the CFG flattening transformation. Indeed, each modification of the program counter requires to show that it does not affect the original variables that were present in the program and vice-versa. The proof is crucial in order to show for example that the expressions that were present in the initial program still evaluate to the exact same value.

The control flow of a program that was modified through the CFG flattening transformation is ensured by a switch statement operating over the program counter variable. Thus, it was also necessary to define multiple lemmata pertaining to switch statements. In particular, a call to (flatten pc s n k) will produce a list of cases numbered from n to n+|s|-1. Thus, one can prove that if the value of the program counter pc is either lower than n or greater than n+|s|-1, then the switch statement will not be able to select a case within the list produced by (flatten pc s n k) and it is actually an equivalence.

```
Lemma select_switch_flatten: forall s pc n k ls id,
   0 <= n ->  n+|s| <= Int.max_unsigned ->
   0 <= pc <= Int.max_unsigned ->
   flatten id s n k = ls ->
   (ls[pc] = [] <->  pc<n \/ n+|s| <= pc).
```

Furthermore, flatten appends the list produced by its recursive calls. Thus, a useful lemma that we use in conjunction with the previous one is the following: if one can prove that no corresponding case can be found in the first part `ls1` of an appended list `ls1+ls2`, then finding a case in the appended list amounts to finding it in the second part `ls2` of the appended list.

```
Lemma select_switch_app_right: forall ls1 ls2 pc,
  ls1[pc]=[] -> (ls1++ls2)[pc]=ls2[pc].
```

7. Implementation and Experiments

Our formal development consists of about 2100 lines of specifications and 6000 lines of proofs. It is integrated into the latest version (i.e. version 2.5) of the CompCert compiler [18]. Our proof relies on the Clight semantics of CompCert and reuses many auxiliary lemmata already present in CompCert and not mentioned in this paper.

Our CFG flattening transformation does not exactly produce the same result as in Figure 1. Indeed, as explained previously (see Section 5.1), some extra switch cases that only increase the program counter were introduced in order to facilitate the proof. Another advantage of this approach is that it differentiates our obfuscation from similar ones, and thus makes it more difficult to detect during reverse engineering (i.e. we generate a more complex CFG). A drawback is that it increases the size of the generated code.

Furthermore, instead of numbering the cases of the `switch` statement from 1 to n as demonstrated by the examples, our transformation is actually parametrized by a numbering function φ over 32-bit machine integers (0 to $2^{32}-1$) and a proof that φ is injective. So the cases are numbered by $\varphi(1),\dots,\varphi(n)$, thus improving the obfuscation of the generated code. Simple examples of such functions are the identity function, shift functions ($n \mapsto n+k \mod 2^{32}$) and the reflection function ($n \mapsto 2^{32}-1-n$).

To evaluate our CFG flattening transformation, we applied it on the set of programs of the CompCert test suite. The size of its larger program is 2233 lines of C code. We also tested our transformation on two common programs that could benefit from program obfuscation: `6pack.c` which is the example compressor program from the FastLZ library, and an implementation of the LZW algorithm [21] that is widely used in many programs such as FFmpeg or LibTIFF.

Table 1 records the computation times necessary to execute programs before and after the CFG flattening transformation. We ran our experiments on a MacBook Pro with a 2,5GHz i5 processor and 8GB RAM. As always with non-trivial obfuscations, our CFG flattening increases the size of the program and thus slows down its execution. Table 1 shows also the slowdown ratio between both programs.

Moreover, we compared our CFG flattening with the CFG flattening implemented in Obfuscator-LLVM, an obfuscator integrated into the LLVM compiler and freely available [11]. Obfuscator-LLVM is not formally verified and operates over the LLVM intermediate representation, that is a lower-level language than Clight. There is a noticeable slowdown in the execution time after the obfuscation with Obfuscator-LLVM as well. Our slowdown ratio ranges from 1.2 to 13.1, and it ranges from 1.1 to 24.6 for Obfuscator-LLVM.

When running our first experiments, we realized that the performance of our obfuscator was not as good as expected, due to a number of `skip` statements that are generated by CompCert before our obfuscation pass. Indeed, the first pass of CompCert translates CompCert C programs into Clight programs by making them deterministic: it operates over non-deterministic programs and pulls side-effects out of expressions and fixes an evaluation order. The correctness proof of this pass is the most challenging one among CompCert proofs. A trick used to facilitate this proof is the use of `skip` statements to materialize evaluation steps of non-deterministic C expressions.

As a consequence, this pass generates at least one `skip` statement for each expression of the initial program. These `skip` statements do not change the performance of the generated assembly code, as they do not correspond to any assembly instruction. However, they are transformed by our obfuscator, which slowdowns the performance of the assembly generated code. To overcome this problem, we added and formally verified in Coq a pass that removes these `skip` statements before our obfuscation pass. More precisely, it transforms all `skip;s` sequences of statements into `s`. Its correctness proof also relies on a simulation diagram involving a matching relation between program states that is much simpler that the `match_states` relation we defined in this paper.

In Table 1, the two columns numbered (4) and (5) correspond to the execution of the `skip`-elimination pass followed by our obfuscation. The last column in Table 1 shows the slowdown ratio between Obfuscator-LLVM and our obfuscator. The results are very encouraging as Obfuscator-LLVM is not formally verified. Our obfuscator is better when this ratio is greater than one. This is the case for 14 of our 24 test programs. Among the 8 remaining programs, our obfuscator has similar execution times on 3 programs, and is slower on 5 programs. The slowest program is only about four times slower. We think that these discrepancies are mainly due to the fact that the obfuscations of Obfuscator-LLVM are performed over a lower level language than Clight, and thus after the optimization passes of the LLVM compiler.

8. Related Work

CFG flattening first appears in Chenxi Wang's Ph.D. thesis [20]. It has since become a widely used obfuscation technique, that is implemented in several advanced obfuscators, including commercially available obfuscators operating over C programs (e.g. [7, 8, 11, 16]). None of these tools is formally verified.

The idea of proving the correctness of obfuscation transformations was first mentioned by Drape *et al.* in [10], where the authors present a framework to specify and prove the correctness of imperative data obfuscations (mainly variable renaming, variable encoding and array splitting). This work formalizes basic obfuscations that do not change the control flow of programs, and operate over a toy imperative language defined by a big-step semantics. Moreover, the correctness proof is a refinement proof, and it is only a paper-and-pencil proof.

To the best of our knowledge, the only work that presents a mechanized proof of code obfuscation transformations is [4]. The authors define and prove correct a few basic obfuscations using the Coq proof assistant. The obfuscations operate over a toy imperative language defined by a big-step semantics. The proof of correctness relies on non-standard semantics called distorted semantics. The main idea of this work is to show that it is possible to devise from the correctness proofs a qualitative measure the potency of these obfuscations.

Our work is integrated into the CompCert formally verified compiler [12] and operates over the Clight language of CompCert. It is a realistic compiler that relies on different intermediate languages and program transformations. The first formally verified transformation operating over Clight was the first the compiler pass (of the very early CompCert compiler) generating Cminor programs [6]. This pass was redesigned since and split into 2 passes relying on a new intermediate language between Clight and Cminor. Since the publication of this work 9 years ago, the Clight language and its semantics became more complex [14].

		CompCert	no skips + obf.		Obfuscator-LLVM		Ratio
Program (1)	LoC (2)	Original (3)	Obfuscated (4)	Ratio (5)	Obfuscated (6)	Ratio (7)	LLVM / no skips + obf. (9)
aes.c	1453	1.015	3.256	3.207	2.290	2.256	0.703
almabench.c	351	0.452	0.781	1.727	0.600	1.327	0.768
binarytrees.c	164	5.001	6.007	1.201	5.387	1.077	0.896
bisect.c	377	4.675	10.127	2.166	24.893	5.324	2.457
chomp.c	368	1.393	4.308	3.092	4.654	3.340	1.080
fannkuch.c	154	0.265	3.306	12.475	6.504	24.543	1.967
fft.c	191	0.095	0.161	1.694	0.302	3.178	1.876
fftsp.c	196	0.001	0.002	2.000	0.004	4.000	2.000
fftw.c	89	2.059	17.817	8.653	6.419	3.117	0.360
fib.c	19	0.164	0.395	2.408	0.662	4.036	1.676
integr.c	32	0.052	0.167	3.211	0.148	2.846	0.886
knucleotide.c	369	0.080	0.152	1.900	0.138	1.725	0.907
lists.c	81	0.386	5.047	13.075	1.214	3.145	0.240
mandelbrot.c	92	1.302	5.559	4.269	24.173	18.566	4.349
nbody.c	174	4.981	17.062	3.425	17.489	3.511	1.025
nsieve.c	57	0.113	0.548	4.849	1.497	13.247	2.731
nsievebits.c	76	0.101	0.389	3.851	0.929	9.198	2.388
perlin.c	75	8.241	32.876	3.989	53.520	6.494	1.627
qsort.c	50	0.293	1.342	4.580	2.703	9.225	2.014
sha3.c	2233	5.202	34.601	6.651	8.321	1.599	0.240
6pack.c	1181	0.843	9.762	11.580	25.583	30.348	2.621
lzw.c	338	21.986	43.065	1.959	segfault	segfault	segfault

Table 1: Benchmark results: execution times (in seconds) of original and obfuscated C programs (using our obfuscator and Obfuscator-LLVM). The last column shows the slowdown ratio between Obfuscator-LLVM and our obfuscator (the higher the better).

9. Conclusion

We presented an obfuscation transformation formally verified in Coq. It relies on a simulation proof involving non-trivial matching relations between semantic states, and operating over a realistic language that is very close to C. Our obfuscation performs CFG flattening on C programs. The experimental results show that it can be applied to realistic programs, while the formal proof ensures that the obfuscation preserves the semantics of programs.

We are currently working on the possibility of restricting the scope of CFG flattening (i.e. obfuscate only parts of the initial program). This doubles the size of the proof (since any statement may be either obfuscated or kept identical), without making it more complex. As further work, we would like to formally verify other basic obfuscations such as those presented in [4] for a toy language. The random combination of these different obfuscations together with the possibility of restricting the scope of CFG flattening would benefit to our CFG flattening and make it much more difficult to detect during reverse engineering.

References

[1] URL `http://www.irisa.fr/celtique/ext/cfg-flatten`. Companion website.

[2] A. W. Appel and S. Blazy. Separation Logic for Small-step Cminor. In Springer-Verlag, editor, *20th Int. Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21, Kaiserslautern, Germany, Sept. 2007.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18. Springer-Verlag, 2001. ISBN 3-540-42456-3.

[4] S. Blazy and R. Giacobazzi. Towards a formally verified obfuscating compiler. In C. Collberg, editor, *SSP 2012 - 2nd ACM SIGPLAN Software Security and Protection Workshop*, Beijing, China, June 2012. ACM SIGPLAN.

[5] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[6] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[7] C. Collberg. The tigress C diversifier/obfuscator, 2014-2015. URL `http://tigress.cs.arizona.edu/`.

[8] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Software Security Series. Addison-Wesley, 2009.

[9] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Trans. Software Eng.*, pages 735–746, 2002.

[10] S. Drape, C. D. Thomborson, and A. Majumdar. Specifying imperative data obfuscations. In *Information Security, 10th International Conference, ISC 2007, Valparaíso, Chile, October 9-12, 2007, Proceedings*, pages 299–314, 2007.

[11] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In B. Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[12] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[13] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[14] X. Leroy. Mechanized semantics for compiler verification. In R. Jhala and A. Igarashi, editors, *Programming Languages and Systems, 10th Asian Symposium, APLAS 2012*, volume 7705 of *Lecture Notes in Computer Science*, pages 386–388. Springer, 2012.

[15] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Automated Reasoning*, 41(1), 2008.

[16] C. Liem, Y. X. Gu, and H. Johnson. A compiler-based infrastructure for software-protection. In *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '08, pages 33–44, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-936-4.

[17] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.

[18] The CompCert development team. *The CompCert formally verified compiler*. Inria, 2008-2015. URL `http://compcert.inria.fr`. Version 2.5.

[19] The Coq development team. *The Coq proof assistant reference manual*. Inria, 2012. URL `http://coq.inria.fr`. Version 8.4.

[20] C. Wang. *A security architecture for survivability mechanisms*. PhD thesis, University of Virginia, 2001.

[21] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.