# On Verifying Secret Control Flow Elimination

**David Knothe**
FZI Research Center for Information Technology, Karlsruhe, Germany

**Oliver Bringmann**
University of Tübingen, Germany
FZI Research Center for Information Technology, Karlsruhe, Germany

**Abstract**

Many countermeasures against timing side-channel attacks have been developed in recent years, including tools to verify that code or a binary is constant-time, compilers or languages that compile into constant-time code, and a formal verification of a compiler that retains the constant-time property.

We take a first step toward formally verifying a C compiler that eliminates control-flow-induced timing side channels. Specifically, we extend CompCert with Partial Control-Flow Linearization (PCFL) [14], a global if-conversion algorithm that was repurposed by Soares et al. [19] for removing timing side channels.

Our transformation is split into multiple steps, separating linearization from instruction predication. One of the intermediate states contains the current program points before and after linearization simultaneously and we exploit a postdominance relation between those to show semantic preservation. We give a new proof that PCFL leaves uniform program points untouched and use it to show that our transformation correctly eliminates all secret control flow.

Although our transformation currently only supports a subset of C, making it unsuitable for use in production, it gives an insight into how a global graph-based linearization technique like PCFL can be verified in CompCert and thereby shows the challenges and obstacles of this undertaking.

**2012 ACM Subject Classification** Security and privacy → Logic and verification; Software and its engineering → Compilers

**Keywords and phrases** CompCert, small-step, linearization, side-channels, constant-time, verification, security, taint analysis

**Digital Object Identifier** 10.4230/LIPIcs.ITP.2025.31

**Supplementary Material** *Software (Source Code)*: `https://github.com/knothed/CompCert-ct` [10]
archived at `swh:1:dir:521e7528bd5a40751111337e04cc57d469e0dd6b`

**Funding** This work was funded within the project FreeSBee by the German Federal Ministry of Research, Technology, and Space with the funding ID 16KIS1737. The responsibility for the content remains with the authors.

## 1 Introduction

Timing side-channels are still a relevant threat for implementations of cryptographic algorithms, nearly 30 years after their discovery by P. Kocher [11]. Many measures against them have been explored and new ones are still developed. For example, there are numerous tools that validate whether a given binary is constant-time like `ct-verif` [2], `dude-ct` [17] or [5]. Measures that automatically remove timing side-channels from arbitrary code are also effective. For example, compiler passes like `Constantine` [6] and `SC-Eliminator` [21] try to find and fix both control-flow- and data-flow-induced side channels. `FaCT` [7] is a DSL which serves a similar purpose. Soares et al. [19] use Partial Control-Flow Linearization (PCFL) [14], a global if-conversion technique, to remove all secret-dependent control flow and thereby only linearize as few branches as possible.

However, while combining a constant-time compiler pass with a subsequent verification that the result is actually constant-time gives security guarantees, it does not show that the transformed code behaves semantically equivalent to the original code. It is still a young field of research to combine formally verified compilation with removing timing side-channels. Besson et. al. [16] give a type-directed, verified constant-time transformation pass for Jasmin, a language specifically designed for verifiable cryptography [1].

CompCert is a formally verified C compiler that is powerful enough to be used in industrial applications [8, 12]. All of its transformations are proven semantically correct, thereby virtually eliminating the possibility of compiler-introduced bugs. Barthe et al. [3] show that a mildly modified version of CompCert compiles programs that are already constant-time down to assembly code that is also constant-time – which is not at all obvious as constant-time is often lost precisely because of compiler optimizations.

This work takes a first step in the direction of formally verifying a CompCert transformation that removes timing leaks from C code. Our transformation:

- removes secret-dependent *control flow*, but does not consider data flow like cache or micro-architectural effects, and
- only works on a subset of C, currently disallowing memory instructions, loops and function calls.

Our work shows that even under these restrictions verifying such a transformation is a complex and effortful endeavor: we have added a total of 9 000 LOC to CompCert. Beneath implementation and verification of four transformations, they contain formalization of graphs, postdominance, influence regions and topological sorting.

Our contributions are as follows:

- We formally verify a compiler transformation that removes secret-dependent control flow for a subset of C, laying the groundwork for prospective verification efforts.
- We describe how global properties, especially postdominance, are established during the transformation, are present in our graph-based intermediate representations and are then used to prove semantic preservation.
- We prove that our transformation correctly integrates with our taint analysis and removes all secret branches, thereby giving a new proof for a uniformity theorem of Hack and Moll [14].

This paper proceeds as follows: Section 2 introduces Partial Control-Flow Linearization and the CompCert compiler. Section 3 explains our multi-step transformation and the intermediate representations that we created, and proves semantic preservation. In Section 4 we show that our transformation actually removes secret control flow and Section 5 sketches how our imposed restrictions can be significantly relaxed. Section 6 presents a new proof of the key theorem about uniformity and Section 7 finally discusses future work and how the imposed restrictions may or may not be relaxed even further.

## 2 Foundations

### 2.1 Partial Control-Flow Linearization

In their work that was originally aimed at auto-vectorization, Hack and Moll [14] distinguish between *uniform* and *varying* variables and conditions and they show that PCFL linearizes all branches that depend on varying conditions. Soares et al. [19] repurpose the context of this transformation to timing side-channel mitigation simply by replacing *varying* with *secret*.

```
if (secret):
  if (public):
    x = cos(x)
  else:
    x = tan(x)
else:
  y = y+1
  x = x+1
return x*y
```

**(a)** Original code.

```
if (public):
  x = secret ? cos(x) : x
else:
  x = secret ? tan(x) : x
y = secret ? y : y+1
x = secret ? x : x+1
return x*y
```

**(b)** Partially linearized code.

**Figure 1** Example linearization. Figure 3 illustrates the control-flow graph of this function.

PCFL in itself is a graph transformation working on a function's control flow graph, but it leaves the details of instruction predication to the implementer of the transformation. A basic example of partial linearization is shown in Figure 1: the secret condition is removed and affected assignments are replaced with select operations, which are assumed to be constant-time. Figure 3 shows how PCFL changes the control flow graph of this code.

## The Algorithm

The main idea of PCFL is that it may change the target of any edge $v \to_E w$ of the CFG but when doing so – thereby going on a detour – it is guaranteed that the original target postdominates the new one. This ensures that all code that was executed in the original graph is also executed after linearization.

PCFL takes an acyclic graph $(V, E)$ with a topological sort[1] $idx \in Sym(V)$. The graph is assumed to have a unique $entry \in V$ and $exit \in V$. Further, a function $secret\_cond : V \to bool$ is required that decides whether the condition at a given vertex is secret.[2]

PCFL returns a transformed graph $(V, E^{lin})$ with the following properties:

- $idx$ is also a topological sort for $(V, E^{lin})$ and $entry$ and $exit$ are also entry and exit vertices of $(V, E^{lin})$
- Each secret condition was linearized, that is, has a single successor in $(V, E^{lin})$
- A map $detour : V \to V \to V$, such that
- $detour\_spec$ holds: for each edge $v \to_E w$ there is an edge $v \to_{E^{lin}} (detour\ v\ w)$ such that $w$ postdominates $(detour\ v\ w)$ in $(V, E^{lin})$.

The algorithm, shown in Algorithm 1, works as follows: We go through the vertices in topological order, at each step considering the vertex $b := idx[i]$. For each edge $b \to s$, we choose the topologically first successor $next$ from either $s$ or the set of deferred edges of $b$. If $b$ is tainted, all edges $b \to s$ get the same successor. Then, $b \to next$ is added to the linearized graph $E_{lin}$, and $detour\ b\ s := next$. To guarantee that $s$ postdominates $next$, we add $next \to s$ to the set of deferred edges $D$.

To show $detour\_spec$, we begin by showing the following basic invariants, where $b = idx[i]$ and $D_i$ and $E_i^{lin}$ denote the respective values at the beginning of the $i$'th iteration:

- $(v, w) \in D_i \implies b \leqslant_{idx} v <_{idx} w$
- $(v, w) \in E_i^{lin} \implies v <_{idx} b \ \wedge\ v <_{idx} w$

[1] We do not require the topological sort to be loop- or dominance-compact, and we do not discuss here how PCFL handles loops.
[2] This function is the result of a taint analysis, see Section 4.

**Algorithm 1** Partial linearization algorithm. The linearized graph is $(V, E^{lin})$.

```
 1: $D := \emptyset, E^{lin} := \emptyset$
 2: for $i$ from 0 to $|V| - 1$ do
 3:     $b := idx[i]$
 4:     $T := \{s \mid (b \to s) \in D\}$
 5:     if $secret\_cond\ b$ then
 6:         $S := \{s \mid (b \to s) \in E\}$
 7:         $next := \min_{idx}(T \cup S)$
 8:         $detour\ b\ s := next\ \forall\ s \in S$
 9:         $E^{lin} := E^{lin} \cup \{(b \to next)\}$
10:         $D := D \cup \{(next \to t) \mid t \in (T \cup S) \setminus \{next\}\}$
11:     else
12:         for each $(b \to s) \in E$ do
13:             $next := \min_{idx}(T \cup \{s\})$
14:             $detour\ b\ s := next$
15:             $E^{lin} := E^{lin} \cup \{(b \to next)\}$
16:             $D := D \cup \{(next \to t) \mid t \in (T \cup \{s\}) \setminus \{next\}\}$
17:         end for
18:     end if
19:     $D := D \setminus \{(b \to s) \mid (b \to s) \in D\}$
20: end for
```

Here, $v <_{idx} w$ means that $v$ appears before $w$ in *idx*, and $v \leqslant_{idx} w$ means $v <_{idx} w \lor v = w$. We can then show the following lemma captures the relationship between $D, E^{lin}$ and *detour*:

▶ **Lemma 1.** *For any $i, v$ with $b = idx[i]$ and $b \to_E v$:*

$$\left[v = detour\ b\ v \ \land\ (b, v) \in E^{lin}_{i+1}\right] \lor \left[(b, detour\ b\ v) \in E^{lin}_{i+1} \ \land\ (detour\ b\ v, v) \in D_{i+1}\right].$$

We also prove Lemma B.3 of [14]: $(v, w) \in D_i \implies w$ postdominates $v$ in $(V, E^{lin})$. Together with the fact that $E^{lin}_i \subset E^{lin}_{i+1}$ we can conclude *detour_spec*, which is present in our development as `Theorem new_target_spec`.

## 2.2 CompCert

The CompCert compiler consists of a handful of intermediate representations (IRs). Each transformation in CompCert transforms a program from one IR into another IR, thereby proving that the program behaviors are similar.[3] This proof is done via small-step simulation diagrams [13]. In small-step semantics, a program executes by making steps between program states until reaching a final state. A state may consist of registers, memory, call stack, the current operation and so on.

To show a simulation diagram between an original and a transformed program, we show that one step of execution in the original IR corresponds to one or more steps of execution in the transformed IR and that the corresponding states of both IRs *match*. An example simulation diagram is shown in Figure 4. Given a few additional properties of the IR's semantics, such a forward simulation diagram yields the required semantic preservation of the transformation [18]. Since all of CompCert's transformations are semantics-preserving, the entire compiler pipeline is as well.

[3] Actually, the behavior of the transformed IR may improve upon the original IR when it has undefined behavior or goes wrong.

A prominent IR is `RTL`, resembling a control flow graph. It separates the program into functions, each one having instructions that operate on a pseudo-register level. Our transformation is designed to work on `RTL`.

▶ Note 2 (CompCert Notation).

- When considering a specific IR, say, `PredRTL`, we write $s_1 \longrightarrow_{pred} s_2$ or $s_1 \longrightarrow^*_{pred} s_2$ for valid transitions or paths between two states. In addition to respecting the graph structure of the IR, these execute the instruction at the given point and update the state accordingly.
- $code@v$ refers to the *instruction* at program point $v \in V$. This instruction is either an *operation* that calculates and assigns a value to a (pseudo-)register (written as `dst <- op(args)`), or a *condition* that evaluates a comparison and branches the control flow depending on its result (written as `if cond(args)`). There are also *nop* and *return* instructions which we do not direct much attention to here because they are uninteresting.
- With *registers* we mean register locations, denoted by *reg*, while a *register state* is a mapping from registers to values, $reg \to val$.

## 3 Transformation And Semantic Preservation

We now describe the transformation and sketch the proof of semantic preservation. For the rest of this paper, fix a function that we want to transform and let $g_{orig} = (V, E_{orig})$ denote its control flow graph.

There are three main restrictions on the function:

- $g_{orig}$ must be acyclic, that is, free of loops.
- The function cannot call other functions[4] and cannot read or write memory – only operations (like `add`, `neg`, `mov` etc.) and conditions are allowed.
- All operations used in the function must be *safe* as we will discuss in Section 3.3.

While these restrictions sound severe, we show in Section 5 that safe operations are in fact only required at certain *non-uniform* program points, and Section 7 discusses how the restrictions on loops, function calls and memory may be lifted in the future.

▶ Note 3 (Notation).

- We fix a topological sort *idx* which is a permutation of $V$. We assume $g_{orig}$ to have a unique[5] $entry \in V$ and $exit \in V$.
- $v \rhd_g w$ denotes that $w$ postdominates $v$ in $g$, that is, every path $v \to^*_g exit_g$ contains $w$. In particular, $v \rhd v$.

$$\begin{tikzcd}
\texttt{RTL} \arrow[r, "\substack{\text{explicit}\\\text{graph}}"] & \texttt{GraphRTL} \arrow[r, "\text{PCFL}"] & \texttt{PredRTL} \arrow[r, "\text{predication}"] & \texttt{SelRTL} \arrow[r, "\substack{\text{select}\\\text{lowering}}"] & \texttt{RTL} \\
 & g_{orig} & (g_{pcfl}, g_{orig}) & g_{pcfl} &
\end{tikzcd}$$

Figure 2 Transformation pipeline.

[4] Our transformation is of course proven correct for every function fulfilling these requirements, even if we cannot exit the main function when compiling real code.

[5] A unique exit can easily be simulated with *gotos* to a shared return statement.

Our work transforms an `RTL` function back to an `RTL` function and proceeds in four steps as displayed in Figure 2: we first equip `RTL` with an explicit graph structure without changing any semantics.[6] Then, PCFL is performed, transforming the graph $g_{orig}$ into $g_{pcfl}$, but leaving the code as is. During predication we get rid of $g_{orig}$ and predicate all operations using select instructions. The last step allows to lower these selects into architecture-specific operation sequences.

## 3.1 `PredRTL`

`PredRTL` is the principal intermediate representation of our transformation: it captures the essence of PCFL by considering both graphs $g_{orig}$ and $g_{pcfl}$ and the current positions in both of them simultaneously and relates them using *detour_spec*.

▶ **Definition 4** (function)**.** A `PredRTL` function[7] consists of:

- $g_{orig}$ and $g_{pcfl}$: the graphs before and after PCFL,
- $detour : V \to V \to V$
- $detour\_spec : \forall v \underset{orig}{\to} w : v \underset{pcfl}{\to} detour\ v\ w \ \wedge\ detour\ v\ w \underset{pcfl}{\triangleright} w.$

When PCFL begins a detour by changing the edge $v \to_{orig} w$ into $v \to_{pcfl} detour\ v\ w$, for semantic preservation it is required that the instructions at $v$ and $w$ are executed while all instructions on the detour are silent, that is, do not change the register state.[8]

Therefore, a `PredRTL` state can convey the information whether it is currently on a detour or not by simply storing the current positions in both $g_{pcfl}$ *and* $g_{orig}$. The latter represents the original path that would be taken through $g_{orig}$ while the former also contains any detours made in $g_{pcfl}$. Only when both positions agree, the current operation is allowed to change register values. We see that there is no need to construct a boolean expression for the predicate – a simple integer comparison is enough.

▶ **Definition 5** (state)**.** We denote a `PredRTL` state by $(v, v_{orig}, rs)$ where $v, v_{orig} \in V$ are the positions in $g_{pcfl}$ and $g_{orig}$, respectively, and $rs : reg \to val$ is the current register state.

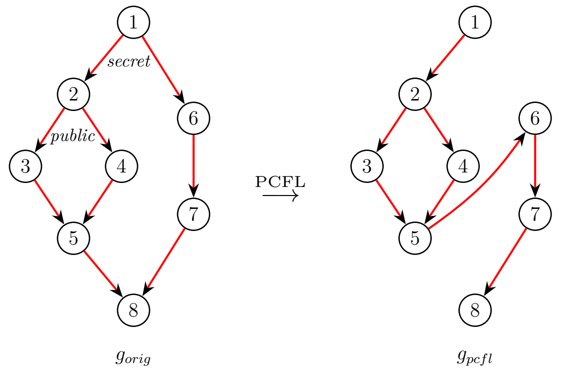
If $v = v_{orig}$, we call the state *live*, otherwise we call it *dummy*.

The initial state is $init := (entry, entry, rs_{params})$ with $rs_{params}$ coming from the arguments passed to the function.

▶ **Definition 6** ($\longrightarrow_{pred}$)**.** The following table defines the valid transitions between `PredRTL` states, depending on whether the source state is live and on the instruction at the program point. The first line inside each cell shows the target state of the transition.

| | **Instruction type** | |
|---|---|---|
| **Source State ↓** | $code@v =$ `dst <- op(args)` with $v \to_{orig} w$ | $code@v =$ `if cond(args)` with $v \xrightarrow{t}_{orig} w_t$ and $v \xrightarrow{f}_{orig} w_f$: |
| Live: $(v, v, rs)$ | $\longrightarrow_{pred} (detour\ v\ w, w, rs')$; $rs' := rs[dst \leftarrow$`eval op(args)` $]$ | $\longrightarrow_{pred} (detour\ v\ w, w, rs)$; $w := ($`eval cond(args)`$)?w_t : w_f$ |
| Dummy: $(v, v_{orig}, rs)$ | $\longrightarrow_{pred} (detour\ v\ w, v_{orig}, rs)$; `op(args)` evaluates successfully | $\longrightarrow_{pred} (detour\ v\ w, v_{orig}, rs)$; $w := ($`eval cond(args)`$)?w_t : w_f$ |

[6] An explicit `Graph` type is handy for constructing and reasoning about graphs independently of instructions.
[7] For clarity, we illustrate a slightly simplified version of all IRs in this paper.
[8] Because we do not allow memory instructions, only the register state is relevant for semantic preservation.



Two edges were redirected: `detour 1 6 = 2` and `detour 5 8 = 6`. For all other edges $v \rightarrow_{orig} w$, `detour v w = w`. The paths $1 \rightarrow 6 \rightarrow 7 \rightarrow 8$ and $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 8$ in `GraphRTL` are translated into $(1,1) \rightarrow (2,6) \rightarrow [(3,6)$ or $(4,6)] \rightarrow (5,6) \rightarrow (6,6) \rightarrow (7,7) \rightarrow (8,8)$ and $(1,1) \rightarrow (2,2) \rightarrow (3,3) \rightarrow (5,5) \rightarrow (6,8) \rightarrow (7,8) \rightarrow (8,8)$, respectively.

**Figure 3** Example translation of a `GraphRTL` function and some paths into `PredRTL`. For simplicity, the register state is omitted from the states.

We see that, while dummy states do not change the register state, we still require that the instruction at a dummy state executes successfully. We will discuss this in more detail in Section 3.3. Using *detour_spec*, we see:

▶ **Corollary 7.** *For any step* $(v, v_{orig}, rs) \longrightarrow_{pred} (w, w_{orig}, rs')$ *we have* $v \rightarrow_{pcfl} w$ *and either* $v_{orig} = w_{orig}$ *or* $v_{orig} \rightarrow_{orig} w_{orig}$. ◀

The following postdominance-relation guarantees that any detour that is taken in $g_{pcfl}$ will eventually come back to its original target:

▶ **Definition 8.** A `PredRTL` state $s$ is *reachable* if there is a path $init \longrightarrow^{*}_{pred} s$.

▶ **Lemma 9** (Postdominance). *For every reachable state* $(v, v_{orig}, rs)$, $v \underset{pcfl}{\triangleright} v_{orig}$.

The proof is a simple induction over the step, using Corollary 7.

## 3.2 `GraphRTL` $\rightarrow$ `PredRTL` Simulation

Let us consider a `GraphRTL` function with the CFG $g_{orig}$. PCFL generates $g_{pcfl}$ and thereby constructs and proves both *detour* and *detour_spec*, as described in Section 2.1.

A `GraphRTL` state, like `RTL`, only consists of the position in $g_{orig}$ and the register state. Figure 3 shows an example translation of some executions in `GraphRTL` to `PredRTL`.

▶ **Definition 10** (Matching). A `GraphRTL` state $(v, rs)$ and a `PredRTL` state $(w, w_{orig}, rs')$ *match* when:

$$(v, rs) \sim (w, w_{orig}, rs') :\Longleftrightarrow v = w = w_{orig} \ \wedge \ rs = rs'.$$

Notably, only live `PredRTL` states can match a `GraphRTL` state.

To prove semantic preservation, we apply a small-step simulation argument as described in [12]. Therefore, consider a `GraphRTL` step $(v, rs) \longrightarrow_{graph} (w, rs')$. We want to show that $(v, v, rs) \longrightarrow^{+}_{pred} (w, w, rs')$. The simulation diagram in Figure 4 outlines the situation.

$$\begin{tikzcd}
\texttt{GraphRTL} & & \texttt{PredRTL} \\
(v, rs) \arrow[rr, dash, "\sim"] \arrow[ddd] & & (v, v, rs) \arrow[d] \\
& & (detour\ v\ w, w, rs') \arrow[dl, dashed] \arrow[dd, dashed, "*"] \\
& (\bullet, w, rs') \arrow[dr, dashed] & \\
(w, rs') \arrow[rr, dash, dashed, "\sim"] & & (w, w, rs')
\end{tikzcd}$$

**Figure 4** Simulation diagram between `GraphRTL` and `PredRTL`.

The first step is $(v, v, rs) \longrightarrow_{pred} (detour\ v\ w, w, rs')$ and by Lemma 9, every reachable state $(\tilde{v}, w, rs')$ fulfills $\tilde{v} \triangleright_{pcfl} w$, so we *have* to reach the state $(w, w, rs')$ after an unknown number of intermediate steps between dummy states, all of which do not change the register state $rs'$ anymore.

The only caveat is that for all the intermediate steps, the respective operation must be executable. To see how this can fail, consider the code in Figure 5: by linearizing a secret guard condition, a possible division by zero is introduced that could not happen before.

Similarly, in Figure 6, CompCert cannot continue simulation at the inner condition because its operand is undefined. Linearization removes the outer secret condition and therefore introduces undefined behavior when $secret = false$.

By restricting the code to only use a certain subset of *safe* operations we can overcome this issue and therefore show semantic preservation.

## 3.3 Safe Operations and Register-Definedness

```
if (sec > 0):
  x = 1 / sec               x1 = 1 / sec
else:                       x2 = 0
  x = 0                     x = (sec > 0) ? x1 : x2
```

**Figure 5** Code where linearization changes the semantics: removing the secret guard condition `if (sec > 0)` lets the linearized code halt if $sec = 0$.

```
pub = pub << 32
// pub is undef now         pub = pub << 32
if (sec):                   a1 = a + 1
  if (pub):                 if (pub):
    a += 1                    a = (sec) ? a1 : a
```

**Figure 6** Code where linearization changes the semantics: when *sec* is $false$, simulation of the linearized code cannot continue at the undefined public condition `if (pub)`.

In CompCert, evaluating an operation returns an `option val`. This may be `None`, in which case the simulation cannot continue. Similarly, evaluating a condition returns an `option bool`, which also stops the simulation in case of `None`. This happens if one of the values to be compared is `Vundef`, which is a special `val` representing undefinedness.

To be able to show that the simulation successfully runs during a detour, we therefore require three things:

1. the `PredRTL` code is *well typed*
2. the `PredRTL` code only uses *safe* operations
3. during execution of `PredRTL`, all values stored in the registers are *well typed* and *defined* at any time.

▶ **Definition 11** (Safe Operation). A value `val` is *defined* if `val <> Vundef`. An operation is *safe* if executing it with correctly typed and defined arguments yields `Some val` such that `val` is again defined.

Unsafe operations are all those than can halt the code (like division) or introduce undefined behavior (like bit-shifts). These include a lot of operations and we describe in Section 5 how this restriction can be weakened.

Welltyping and definedness of the code is verified during `RTL`-to-`GraphRTL` translation while welltyping and definedness of the register state is proven inductively for each possible `PredRTL` step. To get register-definedness and -welltypedness at the initial state, we simply initialize them all to some value, say zero, in their respective type.

The following three lemmata (which we have so far only proven for `x86` as operations and their safety are architecture-specific) summarize what we introduced in this section and let us conclude with the semantic preservation proof.

▷ Claim 12 (Safe Operation Execution).

```
∀ op args, op_safe op -> welltyped op args -> all_defined args ->
∃ (v: val), eval_operation op args = Some v /\ v <> Vundef.
```

▷ Claim 13 (Condition Execution).

```
∀ cond args, welltyped cond args -> all_defined args ->
∃ (b: bool), eval_condition cond args = Some b.
```

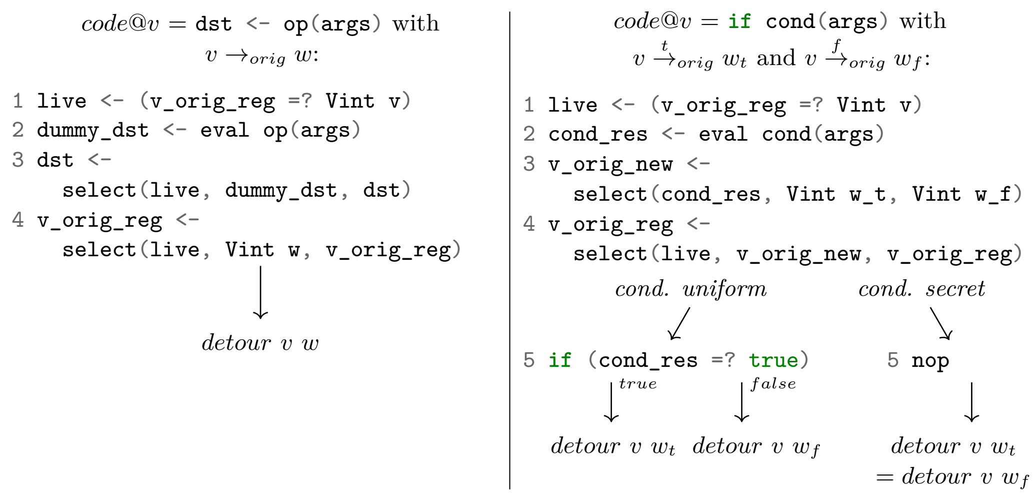
▷ Claim 14 (Welltypedness and Definedness). For every reachable state $(v, v_{orig}, rs)$, we have `rs_welltyped rs` and `rs_defined rs`.

## 3.4 Predication

Predication is the transformation from `PredRTL` to `SelRTL`, which is a similar IR to `GraphRTL`, only augmented by a platform-independent select instruction. During predication, we get rid of the original graph $g_{orig}$ and define both states and transitions between these states only via $g_{pcfl}$. Thereby we rewrite all operations such that conditional operation execution is translated into actual code execution with clearly defined effects.

Like `GraphRTL`, a `SelRTL` state only consists of a program point and register values. We get rid of $v_{orig}$ by instead storing its value in a fresh register called `v_orig_reg` and keeping it updated at each step. A few more fresh registers are used for auxiliary calculations during predication. With `Vint: nat -> val` we inject statically known program points into 64-bit register values.

Figure 7 sketches the predication of operations and conditions. Each instruction is translated into a sequence of instructions: first we determine whether we are at a live state by comparing the program point of the instruction that we translate (which is statically known during the translation) with the current content of the `v_orig_reg` register.


$code@v$ = dst <- op(args) with $v \rightarrow_{orig} w$:
$code@v$ = if cond(args) with $v \xrightarrow{t}_{orig} w_t$ and $v \xrightarrow{f}_{orig} w_f$:
detour $v$ $w$
cond. uniform
cond. secret
true
false
detour $v$ $w_t$
detour $v$ $w_f$
detour $v$ $w_t$ = detour $v$ $w_f$

```
live <- (v_orig_reg =? Vint v)
dummy_dst <- eval op(args)
dst <-
  select(live, dummy_dst, dst)
v_orig_reg <-
  select(live, Vint w, v_orig_reg)
```

```
live <- (v_orig_reg =? Vint v)
cond_res <- eval cond(args)
v_orig_new <-
  select(cond_res, Vint w_t, Vint w_f)
v_orig_reg <-
  select(live, v_orig_new, v_orig_reg)
```

```
if (cond_res =? true)
```

```
nop
```

**Figure 7** Predication of operations and conditions. `r <- select(c,t,f)` either writes the value in register `t` or `f` into `r`, depending on the whether the value in register `c` is *true* or *false*.

- Operations are executed and stored into `dummy_dst` which always succeeds because of the safety measures outlined in Section 3.3. We employ select-operations to write to the operation's actual destination and to `v_orig_reg` depending on whether we are live. The successor of this instruction block stays the successor of the instruction in $g_{pcfl}$.
- Conditions are always evaluated. If we are live, `v_orig_reg` is updated with the new successor in `g_orig`. Only uniform conditions keep their conditional branch while secret conditions are linearized, in which case *detour* $v$ $w_t$ = *detour* $v$ $w_f$ by PCFL.
- Nop instructions do not need to be predicated and neither does the return instruction because it is at the *exit* vertex which can only be reached in $g_{pcfl}$ by a live state.

▶ **Definition 15** (Matching). A `PredRTL` state $(v, v_{orig}, rs)$ and a `SelRTL` state $(w, rs')$ *match* when:

$$(v, v_{orig}, rs) \sim (w, rs') :\Longleftrightarrow v = w \;\wedge\; rs' \geq rs^{9} \;\wedge\; rs'[\texttt{v\_orig\_reg}] = \texttt{Vint}\ v_{orig}.$$

With this matching relation it is not hard, only lengthy, to prove that semantic preservation between `PredRTL` and `SelRTL` holds.

Notice that any `PredRTL` transition $(v, v_{orig}, rs) \longrightarrow_{pred} (w, w_{orig}, rs')$ is translated into a `SelRTL` path $(v, rs_*) \longrightarrow^{+}_{sel} (w, rs'_*)$ and, by Corollary 7, $v \rightarrow_{pcfl} w$. Therefore, a `SelRTL` function does not need to store $g_{orig}$ anymore, only an extended version of $g_{pcfl}$ is required as its CFG.

## 3.5 Select Lowering

To go back from `SelRTL` to `RTL` we maily need to decide how to lower select instructions. We can use the builtin select operation on architectures that have one (which is also what we currently do on `x86`), while on platforms without one we can employ a sequence of bitwise instructions: when the value inside `c` is either 0 or 1, `-c` is either 000...000 or 111...111, so by masking the values in `t` and `f` with `-c`, we can simply translate the instruction `r <- select(c,t,f)` into `r <- ((-c) & t) | ((~(-c)) & f)`.

[9] $rs' \geq rs$ means: $\forall r, rs(r) \neq \texttt{Vundef} \implies rs'(r) = rs(r)$, so $rs'$ may contain additional values.

With the welltypedness and definedness of the registers at hand it should be straightforward to choose the right operations and prove that they give the correct result.

## 4 Control-Flow Security

The transformation preserves semantics irregardless of which conditions are secret. In this section we define our taint analysis – the procedure to determine which registers and conditions are secret – and prove the *control-flow security* of the transformation, meaning that it removes all secret-dependent control flow. In this section we only consider the graph $g_{orig}$ unless specified otherwise; all graph-related notations refer to $g_{orig}$.

### Influence Regions

Given a vertex $c \neq exit$, let $\operatorname{ipd} c$ denote its immediate post-dominator. Let $ir(c)$ be the influence region of $c$ and $ir^+(c)$ its influence region enlarged by $\operatorname{ipd} c$, that is:

$$\begin{aligned} v \in ir(c) &:\Longleftrightarrow c \to^+ v \to^+ \operatorname{ipd} c \\ v \in ir^+(c) &:\Longleftrightarrow v \in ir(c) \;\vee\; v = \operatorname{ipd} c \end{aligned}$$

$ir(c)$ is empty if the instruction at $c$ is no condition, since $c$ then has only one successor.

### 4.1 Taint Analysis

When writing a C function, our development allows the programmer to declare a set of parameters as secret with the `tainted` attribute. The goal of taint analysis is to determine which registers and conditions are secret, that is, may depend on the values of secret parameters during execution. This materializes in two properties, *secret_reg* and *secret_cond*.

▶ **Definition 16** (Uniform). For $v \in V$, let $uni\ v :\Longleftrightarrow \forall c \in V, v \in ir(c) \implies \neg secret\_cond\ c$.

*uni* is equivalent to the *uni* defined by Hack and Moll for which we give a proof in the supplementary material and in our development. If a program point is uniform, it will be reached in $g_{pcfl}$ exactly if it is reached in $g_{orig}$ (see Section 4.2).

We want the following requirements to hold after successful taint analysis:

▷ Claim (T0). For each tainted parameter $p$ we have $secret\_reg\ p$ (as parameters inject into the register space).

▷ Claim (T1). if $code@v =$ `dst <- op(args)` and $any\_secret\ args$, then $secret\_reg\ dst$.

▷ Claim (T2). if $code@v =$ `dst <- op(args)` and $\neg uni\ v$, then $secret\_reg\ dst$.

▷ Claim (T3). if $code@v =$ `if cond(args)` and $any\_secret\ args$, then $secret\_cond\ v$.

Here we used $any\_secret\ regs :\Longleftrightarrow \exists r \in regs, secret\_reg\ r$.

To achieve this, we use the Kildall dataflow inequation solver [9] present in CompCert. We define the state to be a tuple of secret registers and secret conditions. The transfer function works similar to the above requirements, using the current program point's state in place of *secret_reg* and *secret_cond*.

For simplicity, we want a result that is independent of the program point (that is, a register is either always secret or never). We therefore add an extra edge $exit \to entry$ to $g_{orig}$ during the analysis: because this leads to a path between *any* two vertices of the graph (via $v \to^* exit \to start \to^* w$), the Kildall dataflow inequations allow us to prove that the analysis result is the same everywhere, which lets us conclude that Claims T0 to T3 hold.

After above taint analysis has been performed in `GraphRTL`, the `GraphRTL`-to-`PredRTL` transformation uses *secret_cond* to tell PCFL which conditions to linearize.

## 4.2 Proving Control-Flow Security

By *control-flow security of `PredRTL`* we mean that two executions of the same `PredRTL` function that are called with the same public parameters and arbitrary secret ones make exactly the same control flow decisions. This is equivalent to program counter security as defined by Molnar et. al. [15].

To show control-flow security, we need the following *uni-dummy relation* theorem that we will prove in Section 6 and which is equivalent to Theorem 4.1 from [14]:

Uni-Dummy Relation. For every reachable dummy state $(v, v_{orig}, rs)$, we have $\neg uni\ v$. ◁

We now prove control-flow security for a single `PredRTL` step. Let $init_1$ and $init_2$ be two initial states of the function with the same public parameters.

▶ **Definition 17.** $agree\_publicly\ rs\ rs' :\iff \forall r \in reg, (\neg secret\_reg\ r \implies rs(r) = rs'(r))$.

▶ **Lemma 18.**

$$\begin{aligned} \textit{If} \quad & init_1 \longrightarrow^*_{pred} (v', w'_1, rs'_1) \longrightarrow_{pred} (v_1, w_1, rs_1), \\ & init_2 \longrightarrow^*_{pred} (v', w'_2, rs'_2) \longrightarrow_{pred} (v_2, w_2, rs_2)\ \textit{and} \\ & agree\_publicly\ rs'_1\ rs'_2, \\ \textit{then} \quad & v_1 = v_2\ \wedge\ agree\_publicly\ rs_1\ rs_2. \end{aligned}$$

**Proof.** Consider the instruction at $v'$ and whether the states $(v', w'_1, rs'_1)$ and $(v', w'_2, rs'_2)$ are live.

- If *code*@$v'$ = `dst <- op(args)`:
  As $v'$ has only one successor, $v_1 = v_2$ is clear. It remains to show that $rs_1$ and $rs_2$ agree publicly. If both states are dummy, the registers are not updated and so the statement holds by assumption. If both states are live, consider *args*. If *any_secret args*, we have *secret_reg dst* by Claim T1 and so the change to *dst* is irrelevant for public agreement. If $\neg$*any_secret args*, $rs_1(args) = rs_2(args)$ (because *agree_publicly* $rs_1$ $rs_2$) and therefore the result of $op(args)$ is the same in both executions.
  If one state is live and one is dummy, say $v' = w'_1 \neq w'_2$, we make use of the *uni-dummy relation* which yields $\neg uni\ v'$. By Claim T2 it is then *secret_reg dst* and so public agreement is preserved.
- If *code*@$v'$ = `if cond(args)`:
  As the registers are not changed, only $v_1 = v_2$ remains to show. If the condition is uniform, by Claim T3 we have $\neg$*any_secret args* and therefore the result of $cond(args)$ is the same in both executions, so the same branch is taken. If the condition is secret, we use the linearization property of PCFL that secret conditions only have a single successor in $g_{pcfl}$. ◀

We can now conclude control-flow security, which we formulate as follows. Let $s_1 \overset{tr}{\longrightarrow}^*_{pred} s_2$ denote a simulation path from $s_1$ to $s_2$ where $tr$ is the trace of all traversed program points. If two paths have the same trace, it follows that the same sequence of instructions was performed, only with possibly different values.

▶ **Corollary 19** (Control-Flow Security).

$$\textit{If } \ init_1 \overset{tr_1}{\longrightarrow}^*_{pred} (v_1, w_1, rs_1), \ init_2 \overset{tr_2}{\longrightarrow}^*_{pred} (v_2, w_2, rs_2) \ \textit{and } |tr_1| = |tr_2|,$$

$$\textit{then } \ tr_1 = tr_2 \ \wedge \ v_1 = v_2 \ \wedge \ agree\_publicly\ rs_1\ rs_2.$$

**Proof.** We do an induction on $|tr_1| = |tr_2|$. If both are empty, the relevant states are $init_1$ and $init_2$, which both start at the *entry* vertex, so by Claim T0, we get $rs_1 = rs_2$.

Otherwise, remove the most recent program point from $tr_1$ and $tr_2$ to get $tr_1'$ and $tr_2'$. We then have the following situation:

$$init_1 \overset{tr_1'}{\longrightarrow}^*_{pred} (v_1', w_1', rs_1') \longrightarrow_{pred} (v_1, w_1, rs_1)$$
$$init_2 \overset{tr_2'}{\longrightarrow}^*_{pred} (v_2', w_2', rs_2') \longrightarrow_{pred} (v_2, w_2, rs_2)$$

By induction, $tr_1' = tr_2'$, $v_1' = v_2'$ and *agree_publicly* $rs_1'\ rs_2'$, so we can use Lemma 18 to finish the proof. ◀

Our development features control-flow security via `Theorem control_flow_security`.

### Cube Simulation

Now that we have showed the control-flow security of `PredRTL`, it remains to show a similar result for the `SelRTL` and `RTL` functions that are produced by the transformation. This can be done via a cube simulation diagram [3], where one axis represents the two IRs between which we simulate, another one the progress in time and the third one the two different executions of the same function. We have not written this proof in our development but it seems straightforward by doing a case distinction on the possible `PredRTL` steps and employing the control-flow security step lemma from above. It is thereby important to mark all the new registers introduced during predication as secret because they can differ between both executions.

## 5 Allowing Unsafe Instructions

One big restriction of our current development is the limitation to *safe* instructions, as described in Section 3.3, which also includes banning memory instructions because they cannot be predicated.

The *uni-dummy relation* allows us to lift these restrictions considerably. We reformulate its statement: if $v$ is a uniform program point, every reachable state $(v, v_{orig}, rs)$ is live, meaning $v$ is reached in $g_{pcfl}$ exactly if it is reached in $g_{orig}$.

Therefore it is enough to predicate instructions at non-uniform program points, while those at uniform program points can simply be copied to the `SelRTL` code. This allows us to use unsafe operations and memory instructions at all uniform program points.

When doing this, we still have to be careful with the register-definedness: we need to make sure that registers used at *non*uniform program points are defined, otherwise the simulation may break just as described in Section 3.3. So, an unsafe operation can only be allowed when its destination register is not used as the argument of any operation at a nonuniform program point. We therefore cannot fully remove but instead relax the `rs_defined rs` condition to `critical_rs_defined rs` where the set of *critical registers* has been determined together with the taint analysis.

We did not yet implement or prove the critical register relaxation and with it the permission for unsafe operations that we described here in our development. To show a proof of concept however, we added support for the `Iload` operation at nonuniform program points to allow loading from memory and use the *uni-dummy relation* to show that these `Iload`s do not have to be predicated. Because we did not relax `rs_defined`, we cheated a bit by axiomatically requiring that for any `Iload` that returns `Some val`, `val <> Vundef`.

## 6 Proving the Uni-Dummy Relation

The *uni-dummy relation* was prominent in the previous sections as it was required both for proving control-flow security and for restricting predication to fewer instructions. While its statement is similar to Theorem 4.1 in [14] which was already proven there, we will still provide a different proof in this section because our formulation with influence regions provides a slightly different insight than theirs using control dependence.

As in Section 4, all graph-related notations in this section refer to $g_{orig}$.

**▶ Definition 20.**

- Given $c_1, c_2 \in V$, we say that their influence regions *intersect* if

$$c_1 \cap c_2 :\Longleftrightarrow \exists x, x \in ir^+(c_1) \ \wedge \ x \in ir^+(c_2).$$

- Indirect intersection is denoted by an *intersection chain*: $c_1 \cap c_2 \cap \cdots \cap c_{k-1} \cap c_k$.
- For $v, w \in V$, define

$$\begin{aligned} chain\ v\ w :\Longleftrightarrow \exists c_v, c_w \in V : v \in ir^+(c_v) \ \wedge \ w \in ir^+(c_w) \\ \wedge \ c_v \cap \cdots \cap c_w \ \wedge \ \forall c \in \{c_v, \ldots, c_w\}, secret\_cond\ c. \end{aligned}$$

If *chain* $v$ $w$, then $v$ and $w$ are related by a chain of intersecting influence regions, each coming from a secret condition. When PCFL changes an edge target, it relates to the original one via *chain*:

▷ Claim 21. $\forall v \to_{orig} w$ with $w \neq detour\ v\ w$, we have $chain\ w\ (detour\ v\ w)$.

Figure 8 illustrates this: for the new edge $6 \to 7$ there is $6 \in ir^+(1)$, $7 \in ir^+(3)$ and $1 \cap 2 \cap 3$ with all of $1, 2, 3$ being secret conditions.

Assuming Claim 21 we can prove the following theorem that is stronger than the uni-dummy relation:

**▶ Theorem 22.** *For every reachable dummy state $(v, v_{orig}, rs)$, we have chain $v$ $v_{orig}$.*

**Proof.** We prove the statement via induction on $\longrightarrow_{pred}$. Consider the previous step that led to this state:

- $(w, w, rs') \longrightarrow_{pred} (v, v_{orig}, rs)$: this implies $w \to_{orig} v_{orig}$ and $v = detour\ w\ v_{orig}$. The statement follows from Claim 21.

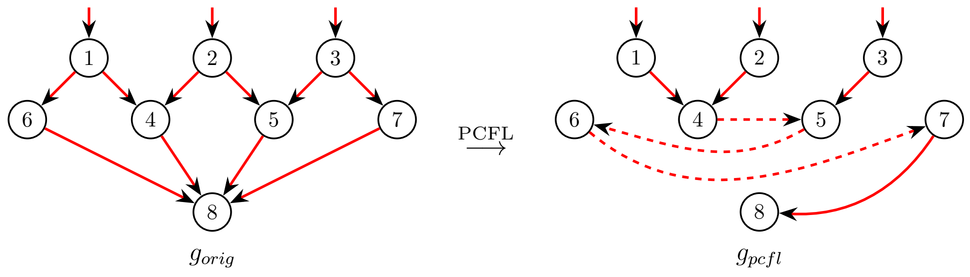


**Figure 8** All conditions 1, 2 and 3 are secret. Dashed edges were created by PCFL. We see that for all new edges $v \to w$ we have *chain* $v\ w$ (influence regions are considered in $g_{orig}$, not in $g_{pcfl}$).

- $(w, v_{orig}, rs') \longrightarrow_{pred} (v, v_{orig}, rs)$ with $w \neq v_{orig}$: this implies $w \to_{orig} w'$ with $v = detour\ w\ w'$. By induction, *chain* $w\ v_{orig}$ and by Claim 21, *chain* $v\ w'$, therefore $c_w \cap \cdots \cap c_{vorig}$ and $c_v \cap \cdots \cap c_{w'}$.
  If $w \neq \operatorname{ipd} c_w$ then by $w \to w'$ we get $w' \in ir^+(c_w)$ and therefore the following is a valid intersection chain for $v$ and $v_{orig}$:

$$c_v \cap \cdots \cap \underbrace{c_{w'} \cap c_w}_{\text{contain } w'} \cap \cdots \cap c_{vorig}.$$

  Otherwise, $\operatorname{ipd} c_w = w <_{idx} v_{orig} \leqslant_{idx} \operatorname{ipd} c_{vorig}$ so we use below Claim 23 to find a $\tilde{c} \in \{c_w, \ldots, c_{vorig}\}$ with $w \in ir(\tilde{c})$ and therefore $w' \in ir^+(\tilde{c})$. The chain is then:

$$c_v \cap \cdots \cap \underbrace{c_{w'} \cap \tilde{c}}_{\text{contain } w'} \cap \cdots \cap c_{vorig}.$$ ◀

▷ Claim 23 (Chain Containment). If $c_1 \cap \cdots \cap c_k$ and $\operatorname{ipd} c_1 <_{idx} \operatorname{ipd} c_k$, then $\exists i, \operatorname{ipd} c_1 \in ir(c_i)$.

A proof for Claim 23 is provided in the supplementary material and in our development. We will now restate and prove the uni-dummy relation, present as `Corollary uni_chain_path` in our development.

▶ **Theorem 24.** *For every reachable dummy state $(v, v_{orig}, rs)$, we have $\neg uni\ v$.*

**Proof.** Let $(v, v_{orig}, rs)$ be an reachable dummy state. By Lemma 22 it is *chain* $v\ v_{orig}$, so $c_v \cap \cdots \cap c_{vorig}$ with $v \in ir^+(c_v)$ and all of $\{c_v, \ldots, c_{vorig}\}$ are secret. To show $\neg uni\ v$, we need a $\tilde{c}$ with $v \in ir(\tilde{c}) \ \wedge\ secret\_cond\ \tilde{c}$. If $c \neq \operatorname{ipd} c_v$, we can just use $c_v$. Otherwise, we use Claim 23, noting that $\operatorname{ipd} c_v = v <_{idx} v_{orig} \leqslant_{idx} \operatorname{ipd} c_{vorig}$, which gives us such a $\tilde{c} \in \{c_v, \ldots, c_{vorig}\}$. ◀

## 6.1 PCFL Invariant

It remains to show Claim 21 which is, in contrast to what we have shown so far, a statement about PCFL itself.

Consider the PCFL algorithm as described in Section 2.1 and let $D_i$ denote the set of deferred edges before the $i$'th step. Recall the invariant from Section 2.1 stating that $(v, w) \in D_i \implies v <_{idx} w$. We now inductively prove the additional invariant: $(v, w) \in D_i \implies chain\ v\ w$. Trivially, $D_0$ is empty.

After processing $b$ – the $i$'th vertex in $idx$ – an edge $(v, w) \in D_{i+1}$ arises in one of five ways:

1. $(v, w) \in D_i$. The invariant holds by induction.
2. $b \to v \ \wedge \ b \to w$ with $b$ a secret condition. As $v \in ir^+(b) \ \wedge \ w \in ir^+(b)$, we can use the trivial chain $b \cap b$.
3. $b \to v \ \wedge \ (b, w) \in D_i$ with $v <_{idx} w$. By induction, there is a chain $c_b \cap \cdots \cap c_w$ with $b \in ir^+(c_b) \ \wedge \ w \in ir^+(c_w)$.
   If $b \neq \operatorname{ipd} c_b$ then $v \in ir^+(c_b)$ and so the same chain works for *chain* $v$ $w$. Otherwise, from $\operatorname{ipd} c_b = b <_{idx} w \leqslant_{idx} \operatorname{ipd} c_w$ and Claim 23 we get a $\tilde{c}$ with $b \in ir(\tilde{c})$, so $v \in ir^+(\tilde{c})$ and so $\tilde{c} \cap \cdots \cap c_w$ gives the required chain.
4. $(b, v) \in D_i \ \wedge \ b \to w$ with $v <_{idx} w$. This case is equivalent to the one above by noting *chain* $v$ $w \equiv$ *chain* $w$ $v$.
5. $(b, v) \in D_i \ \wedge \ (b, w) \in D_i$ with $v <_{idx} w$. By induction, there are two chains $c_b^1 \cap \cdots \cap c_v$ and $c_b^2 \cap \cdots \cap c_w$ and so we get the required chain via:

$$c_v \cap \cdots \cap \underbrace{c_b^1 \cap c_b^2}_{\text{contain } b} \cap \cdots \cap c_w.$$

Proof of Claim 21. To conclude Claim 21, assume $v \to w$ and *detour* $v$ $w =: w' \neq w$:

- Either $v \to w'$ with $v$ secret: then we have $w \in ir^+(v)$ and $w' \in ir^+(v)$, giving *chain* $w$ $w'$.
- Otherwise $(v, w') \in D_i$, giving *chain* $v$ $w'$ by above invariant. We conclude *chain* $w$ $w'$ again using Claim 23 and $v <_{idx} w'$. ◁

## 7 Future Work

This work can be expanded in many ways, most prominently by reducing the restrictions that we have imposed initially. We shortly sketch how this may be done in each case.

### Memory and Unsafe Operations

In Section 5 we already described how the restriction to safe operations and memory instructions can be weakened, allowing us to use them at uniform program points given some requirements on the registers they are allowed to write to.

It is also possible to allow memory writes or unsafe operations at nonuniform program points to a certain degree. For example, a division like `x <- 1/y` may be transformed into a two-step safe division before predication: `y' <- select(y =? 0, 1, y) ;; x <- 1/y'`. Soares et. al [20] describe a similar transformation for memory accesses: a *shadow memory* can be introduced and a memory store like `*x <- y` can be transformed into `p <- select(live, x, shadow) ;; *p <- y` during predication; loads work analogously. While this allows compiling code with arbitrary unsafe and memory instructions (and thereby maintaining verified control-flow security), it introduces cache effects that are not at all obvious to mitigate.

### Function Calls

Function calls at uniform locations with all-public parameters can be handled similarly to unsafe operations as described in Section 5.

If we allow secret parameters in function calls, we need a global taint analysis that guarantees that every parameter of a function that is ever instantiated with a secret value is marked as `tainted`. This global taint analysis may also be outsourced to other tools or manually annotated by the user and then assumed as an axiom in CompCert.

When function calls are additionally allowed at nonuniform locations, we would have to transform them in such a way that they are fully side-effect free (that is, no external calls and only shadow-memory access). Without assuming some kind of acyclicity in the call graph we will not be able to prove semantic preservation because a dummy function call during a detour may never return control back to its caller.

## Loops

Assuming all loops only have uniform exit edges (and are in a nice enough form), it should be manageable to verify PCFL: first do a loop analysis and then either perform PCFL after removing all back-edges and reinsert them afterwards, or transform each loop in itself, each being an acyclic graph without its back-edge. Beyond integrating a verified loop analysis into CompCert (which has already been done in [4]) one would have to explore and reformulate what happens to the postdominance relationship between $v$ and $v_{orig}$ by either restricting attention to a relevant loop-subgraph during simulation or by finding a sensible global formulation. If we require that each loop has a single exit edge, by uniformity of its condition it should be straightforward to show that the loop exit is taken simultaneously in `PredRTL` and in `GraphRTL`.

If we allow loops to have secret exit edges, it is impossible to prove semantic preservation using the extended transformation described by Hack and Moll [14] or by Soares et al. [19]. This is not only because the transformation is complex and extremely intricate, especially when considering nested loops, but because it can break simulation: a loop that exited through a secret exit before may never exit after the transformation. Additional external guarantees (or a weaker notion of simulation that is allowed to diverge after the transformation of a terminating program) and an immense time effort would be required for verifying such a transformation.

## Control-Flow Security

We have only formally proven that `PredRTL`-code is free of secret branches. To maintain this guarantee through the full chain of transformations down to assembly code, one could integrate the work done by Barthe et al. [3] which verifies that most CompCert-transformations are constant-time preserving.

## References

1 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, CCS '17, pages 1807–1823, New York, NY, USA, 2017. ACM. `doi:10.1145/3133956.3134078`.

2 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, SEC'16, pages 53–70, USA, 2016. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida`.

3 Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL):7:1–7:30, December 2020. `doi:10.1145/3371075`.

4 Sandrine Blazy, André Maroneze, and David Pichardie. Formal verification of loop bound estimation for WCET analysis. In *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, volume 8164 of *Lecture Notes in Computer Science*, pages 281–303, Berlin, Heidelberg, 2013. Springer. `doi:10.1007/978-3-642-54108-7_15`.

5 Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, volume 10492 of *Lecture Notes in Computer Science*, pages 260–277, Cham, 2017. Springer. `doi:10.1007/978-3-319-66402-6_16`.

6 Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, CCS '21, pages 715–733, New York, NY, USA, 2021. ACM. `doi:10.1145/3460120.3484583`.

7 Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a DSL for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, PLDI 2019, pages 174–189, New York, NY, USA, 2019. ACM. `doi:10.1145/3314221.3314605`.

8 Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards formally verified optimizing compilation in flight control software. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop PPES 2011, March 18, 2011, Grenoble, France*, volume 18 of *OASIcs*, pages 59–68, Toulouse, France, February 2011. AAAF, SEE, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany. `doi:10.4230/OASIcs.PPES.2011.59`.

9 Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM Press. `doi:10.1145/512927.512945`.

10 David Knothe. CompCert+PCFL. Software, swhId: `swh:1:dir:521e7528bd5a40751111337e04cc57d469e0dd6b` (visited on 2025-07-17). URL: `https://github.com/knothed/CompCert-ct`.

11 Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. `doi:10.1007/3-540-68697-5_9`.

12 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. `doi:10.1145/1538788.1538814`.

13 Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995. `doi:10.1006/inco.1995.1134`.

14 Simon Moll and Sebastian Hack. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, PLDI 2018, pages 543–556, New York, NY, USA, 2018. ACM. `doi:10.1145/3192366.3192413`.

15 David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168, Berlin, Heidelberg, 2005. Springer. `doi:10.1007/11734727_14`.

**16** Gautier Raimondi, Frédéric Besson, and Thomas P. Jensen. Type-directed program transformation for constant-time enforcement. In *International Symposium on Principles and Practice of Declarative Programming, PPDP 2023, Lisboa, Portugal, October 22-23, 2023*, PPDP '23, pages 6:1–6:13, New York, NY, USA, 2023. ACM. `doi:10.1145/3610612.3610618`.

**17** Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, DATE '17, pages 1697–1702, Leuven, BEL, 2017. IEEE. `doi:10.23919/DATE.2017.7927267`.

**18** Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013. `doi:10.1145/2487241.2487248`.

**19** Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. Side-channel elimination via partial control-flow linearization. *ACM Trans. Program. Lang. Syst.*, 45(2):13:1–13:43, June 2023. `doi:10.1145/3594736`.

**20** Luigi Soares and Fernando Magno Quintão Pereira. Memory-safe elimination of side channels. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 200–210. IEEE, 2021. `doi:10.1109/CGO51591.2021.9370305`.

**21** Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, ISSTA 2018, pages 15–26, New York, NY, USA, 2018. ACM. `doi:10.1145/3213846.3213851`.