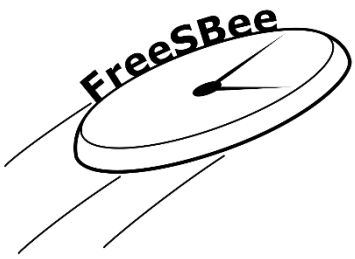


— D3.1 Middle-end IR transformation

Mitigating secret-dependent control-flow and data-flow timing effects

Version: 1.01
Publication date: 27.08.2024
Authors: David Oswald (KaOs)
Sebastian Reiter (FZI)
Julian Ganz (FZI)
David Knothe (FZI)
Anton Paule (FZI)
Simon Wegener (AbsInt)
Stephan Wilhelm (AbsInt)
Contact: David Knothe <knothe@fzi.de>

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

– Table of content

1 Introduction

The aim of the FreeSBee project is the automatic detection and mitigation of timing side-channel attacks in embedded software. A timing side-channel is an attack surface that can arise when the value of secret data (like for example a cryptographic key or a user password) has a measurable influence on the execution time of a program. When an attacker knows the workings of the cryptographic function, they may be able to draw conclusions about the secret data by precisely analysing these variations in execution time over multiple runs. This is not just a theoretical attack but has been demonstrated and exploited many times in the past, as described in our first report [1].

There are two main ways how a program's execution time can depend on the value of data:

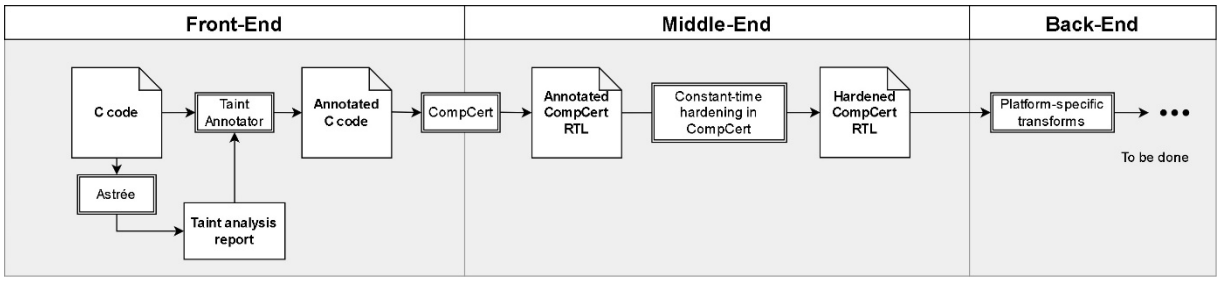
- through data dependence, when operations take different amounts of time depending on the data they operate on (for example through cache effects), and
- through control dependence, when the program performs a different sequence of operations depending on the data (due to secret-dependent conditional statements).

To mitigate the attack surface of timing side-channels, both of these dependencies must be eliminated as far as possible. Therefore, an understanding is required of which data is control- or data-dependent on secret data. The program is then compiled in such a way that all timing variations introduced by any of this secret-dependent data are removed, making the program timing-secure.

The previous report [2] described how FreeSBee employs *taint analysis* to detect secret-dependent values in a program. This report describes the techniques used by FreeSBee to eliminate control dependence and, to some extent, mitigate data dependence of secret values to minimize their influence on program execution time.

1.1 Current Workflow

The below figure shows our current workflow. The frontend, as described in our previous report [2], analyses C code using Astrée [3], further processes it and annotates the C code with custom taint annotations. In the middle-end, a custom version of the CompCert compiler [4] digests these taint annotations and uses them to analyse and transform a platform-independent intermediate representation (RTL) into constant-time code. This transformation is described in the rest of the report.



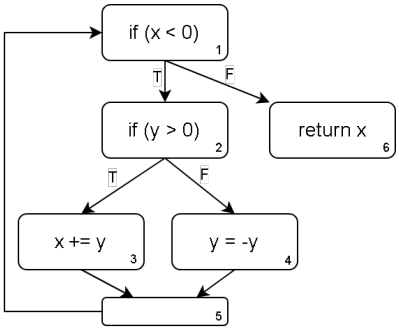
The toolchain for front-end and middle-end.

2 Microarchitecture-independent constant time hardening

2.1 Control-Flow Graphs

A control-flow graph is a standard graph-based representation of a procedure where vertices stand for (sequences of) instructions and edges represent the control-flow. The instructions may be either native processor instructions or more abstract kinds of instructions.

A vertex having two outgoing edges represents conditional jumps. Executing a function step by step is equivalent to tracing a (possibly infinite) path through the control-flow graph, beginning at its entry vertex and performing the instructions at visited vertices while going along conditional edges depending on the evaluated conditions.



An exemplary CFG containing a while-loop and an if-statement.

The control-flow is *secret-dependent* or *tainted* if the condition of some conditional jump depends on a secret value. In terms of the CFG this means that, when executing the function step by step, the path that is taken in the CFG depends on a secret value.

Vice versa, the control flow is *secret-independent* or *clean* when running the CFG with two different sets of inputs that only disagree on their secret values (but agree on the clean values) does not change the taken path – the same sequence of instructions will be executed, just with possibly different data.

Below, we describe the vital task of making control-flow of general functions secret-independent.

2.2 Control-Flow Linearization

The task of control-flow linearization is to remove all tainted control flow. In its most simple form, linearizing an if-statement works by executing both branches of the if-statement, but in such a way that only the results of the “correct” statement are observed while the other one is discarded. This can be done using the ternary operator (a ? b : c) as follows:

```
if (c > 0)
  x += y;
else
  y = y * 3;
```

Original if-statement

```
int d = (c > 0);
x_new = x + y;
x = d ? x_new : x;
y_new = y * 3;
y = d ? y_new : y;
```

Linearized if-statement

Of course, this only has its desired effect when the ternary operator is compiled into a constant-time operation like SELECT. If the ternary operator is compiled into another if-statement (because the target platform doesn't have a SELECT operation), this is useless. Other, more platform-independent ways of achieving the same goal are the following:

```
int d = (c > 0);
x = d * x_new + (1-d) * x;
```

Using a linear combination

```
// d is either 0 or 111...111
int d = -(c > 0);
x = (d & x_new) ^ ((~d) & x);
```

Using bitwise operations

(For these to work correctly, the type of **d** would have to be chosen to match the types of **x** and **y**).

These linearized versions have no conditional jumps in them and therefore have clean control-flow. The only sources for timing leaks in this code may come through operations whose timing depends on the data they operate on, e.g. a multiplication. Mitigating this, however, is the objective of AP4.

This technique can be used to linearize simple, standalone if-statements. However, things get trickier when considering a full control-flow graph that may contain nested if-statements and loops. For this, we use a technique called *Partial Control-Flow Linearization*.

2.3 Partial Control-Flow Linearization (PCFL)

A few years ago, Hack and Moll [5] invented Partial Control-Flow Linearization (PCFL) as an advancement on automatic vectorization. PCFL applies perfectly to our purpose of timing side-channel removal as well, as noticed by L. Soares [6] who implemented it in LLVM.

PCFL linearizes a whole control-flow graph, thereby trying to minimize the number of if-statements that are linearized. Obviously, all tainted if-statements have to be linearized – PCFL avoids linearizing clean ones whenever possible. The naïve approach of simply linearizing *every* if-statement in the function has obvious drawbacks: it can dramatically increase function execution time, it makes loops impossible, and it gets problematic when a function contains side-effects.

We will now outline the workings of PCFL, as described by Hack and Moll [5].

2.3.1 PCFL of an Acyclic CFG

In its simplest form, PCFL works on an acyclic CFG, which can be obtained by removing loop backedges[1] from a reducible CFG – see section 2.3.2 for more details.

PCFL consists of two steps: transforming the graph edges (i.e. removing tainted control flow) and predicating relevant statements (i.e. assuring that the program still behaves the same as before).

2.3.1.1 Transforming the CFG edges

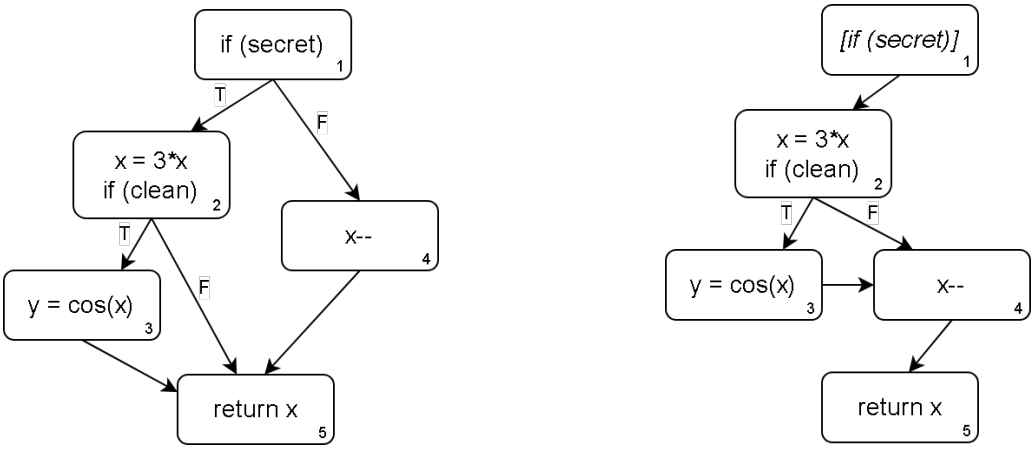
Without loss of generality, we assume that the acyclic CFG has a unique entry vertex (the entry of the function) and a unique exit vertex (the unique return statement). PCFL iterates the vertices in a topological order and does the following to each vertex **v**:

- If **v** ends in a secret-dependent conditional statement: remove the condition at **v** and remove all outgoing edges from **v**. Add a *single* new edge **v** → **x**.
- Otherwise, map each outgoing edge **v** → **w** to an edge **v** → **x**.

[1] As defined by standard terminology, e.g. by LLVM [8]. For example, a backedge is the edge from the latch of the loop (`i++`) to its header (`i<n`).

The key to this transform is that every edge **v** → **w** in the original CFG either stays as is, or is remapped to another edge **v** → **x** with some **x**[2] such that **w** *post-dominates* **x**: this means that, regardless what path is taken starting from **x**, it will come to **w** at some point. Therefore, when an edge is remapped to another target, it is guaranteed that the original target will still be reached, possibly with a few steps in-between.

Because we removed all tainted conditional branches (by collapsing all outgoing edges **v** → $\mathbf{w_1}$, …, **v** → $\mathbf{w_i}$ of a tainted condition into a single one **v** → **x**), the resulting CFG has no more tainted control-flow.



Example of the PCFL graph transform on a simple CFG. On the left, the original graph. On the right, the transformed (but not yet predicated) graph. While the edge 1 → 4 of the secret condition is merged into the edge 1 → 2, it is ensured that 4 post-dominates 2 after the transformation, so both 2 and 4 are reached in every execution. The clean condition is not removed by PCFL however, possibly sparing an expensive call to **cos**.

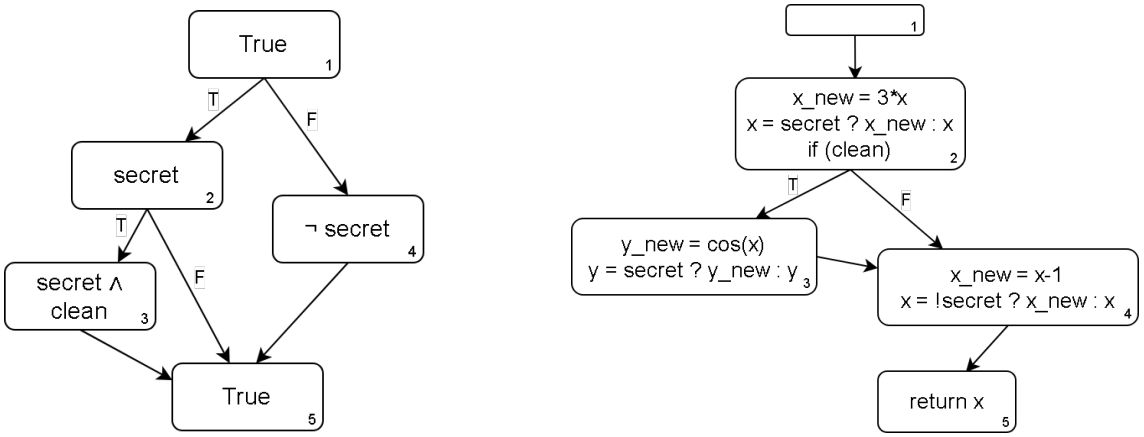
2.3.1.2 Predication and Rewriting

To guarantee that the transformed program still behaves the same as the original one, *predication* has to be implemented correctly. This is what we've seen beforehand: both branches of the if-statement are executed but only one actually has visible impact to the program state.

In PCFL, we do the same. First, we calculate a predicate expression for each vertex by combining the predicates of all its predecessors with their edge conditions – a standard technique. Then, we predicate each assignment in the CFG as shown before, by using the ternary operator for example.

Fortunately, Lemma 4.1 of [5] allows us to cut the number of predications drastically: vertices whose predicate expressions are clean (i.e. do not contain secret variables after being simplified) are *not* affected by PCFL, meaning they do not need to be predicated. This is a great profit of the design of PCFL, as we would have to generate a lot of unnecessary boilerplate code without this locality guarantee of Lemma 4.1. Also, further analyses can be used to reduce the number of instructions required for predication, as can be seen in the example.

[2] How this **x** is determined exactly is described in [5]. Broadly, it works by carefully building a *deferral relation* between vertices – **D v w** meaning that **w** needs to post-dominate **v** in the resulting CFG – and then by choosing the topologically smallest element of all deferred vertices at **v** and all outgoing edges of **v**.



On the left, predication of the original CFG from above; each vertex is assigned a boolean expression in the variables representing the if-conditions. On the right, the predication applied to the transformed graph, giving the final result of PCFL. All relevant statements in the right graph are predicated by the respective predication expression from the left graph. Note that vertex 3 on the right is predicated with **secret**, not with **secret ∧ clean** because analysing the right graph yields that, when vertex 3 is reached, **clean** is always True.

Section 2.4 will give details on how we predicate statements with side effects like memory accesses or function calls.

2.3.2 Linearizing Loops

We require loops to have a unique backedge. The backedges are removed for PCFL and are added afterwards. That this yields a correct result is described in [5].

Each loop may have multiple (but at least one) exit edges[3]. Such an exit edge is always tied to a conditional statement and may therefore either be tainted or clean.

If a loop has only clean exits, nothing additional has to be done. It gets more interesting however if a loop also has tainted exits[4]. Consider the following loop comparing two strings, where the content of **pwd** is tainted (the length **n** is clean however):

```
bool compare(char* pwd, char* str, int n) {
  for (int i=0; i<n; i++)
    if (pwd[i] != str[i])
      return false;
  return true;
}
```

The loop has two exit conditions – **i >= n** and **pwd[i] == str[i]** – the first one being clean and the second one being tainted. The loop takes a different number of iterations depending on the length of the shared prefix of **pwd** and **str**, which is a severe timing leak, allowing an attacker to guess the password character by character. The underlying issue is the tainted condition that exits the loop.

[3] An exit edge is an edge that, when taken, exits the loop.
[4] But not if it *only* has tainted exits: then it is impossible to linearize the loop if we do not know some static upper bound on the number of loop iterations (which would serve as a clean exit condition).

This cannot simply be solved by applying PCFL to the graph with backedges removed as this would destroy the structure of the loop. Instead, we have to preprocess the loop, which is described in more detail in [5] and [6].

Broadly, this works as follows: we only allow the loop to exit through one of its clean exits. If the loop would exit through a tainted exit, we continue loop execution, but in *dummy mode*. When the loop runs in dummy mode, statements that are executed have no effect that is visible from outside the loop. This ensures that the loop continues running until hitting a clean exit but without changing the program's semantics.

```
bool compare(char* pwd, char* str, int n) {
  bool dummy = false;
  for (int i=0; i<n; i++) {
    bool c = pwd[i] != str[i];
    dummy = dummy | c;
  }
  bool result = ~dummy;
  return result;
}
```

Transformed version. The tainted exit of the loop was removed and replaced by the linearized decision of returning true or false *after* the loop.

Because the loop may run in dummy mode, every instruction inside the loop has to be predicated by the loop-private dummy-mode variable. To do this correctly, we have to consider a few things: statements inside a nested loop may only have a side effect if the loop and all parent loops are *not* currently in dummy mode, so each loop must be aware of its parent. Furthermore, it is important to correctly handle values that are updated inside a loop while it is running in dummy mode: these values must be updated during the dummy execution (otherwise the loop may not exit) but have to be reversed after exiting the loop, given they are used again later. For example, the following code reuses **i** after the loop. In the transformed code, **i** continues increasing even when the loop is in dummy-mode (to ensure that we hit the **i>=n** condition), but is reset afterwards (to ensure the correct return value of the function).

```
bool cmp(char* pwd, char* str, int n) {
  int i=0;
  for (; i<n; i++)
    if (pwd[i] != str[i])
      break;
  return (i==n);
}
```

```
bool cmp(char* pwd, char* str, int n) {
  int i=0;
  int i_copy=0;
  bool dummy = false;
  for (; i<n; i++) {
    bool c = (pwd[i] != str[i]);
    dummy = dummy | c;
    i_copy = dummy ? i_copy : i;
  }
  i = i_copy;
  return (i==n);
}
```

Original code on the left and linearized code on the right.

Doing this transform correctly is tricky, especially when considering nested loops. More details about the implementation can be extracted from our artefact[5].

2.4 Handling Complex Instructions

Until now, we described predication of side-effect-free instructions by using the ternary operator or similar means. It is thereby important that the newly created selection statements are both timing-invariant, i.e. do not introduce a new timing leak, and are crash-free, i.e. do not introduce the risk of failing when applied on arbitrary data.

This does not work however for many statements that are used in normal productive code like division operations, memory accesses or function calls. Here we give a few thoughts on how to handle those statements.

2.4.1 Divisions

Consider the following code and its linearization, given that **b** is marked as secret:

```
int safe_div(int a, int b) {
  if (b == 0)
    return 0;
  else
    return a/b;
}
```

```
int safe_div(int a, int b) {
  bool cond = (b == 0);
  int result_1 = 0;
  int result_2 = a/b;
  result = cond ? result_1 : result_2;
  return result;
}
```

We immediately see a problem when the linearized version is executed with **b==0**: the division by zero, which should have been guarded by the if-statement, is now executing nonetheless.

It is however very straightforward to fix this: before the division, **b** can be modified such that it stays unchanged when it is nonzero but gets changed, e.g. to one, if it is zero:

```
int safe_div(int a, int b) {
  bool cond = (b == 0);
  int result_1 = 0;
  int b_ = cond ? 1 : b;
  int result_2 = a/b_;
  result = cond ? result_1 : result_2;
  return result;
}
```

This applies also to similar operations like modulo.

2.4.2 Memory accesses

A similar problem applies to memory accesses, both read and write: accessing memory at a tainted location (i.e. where it would not have been accessed in the original program) may yield an out-of-

[5] Available on the FZI-internal Git repository, tagged with “D3.1”

bounds-access that was not there before the transform. Because we cannot statically detect which memory accesses are valid[6], we have to resort to a similar technique as in the division case, which is inspired by [7].

In particular, we may introduce a new memory location called **dummy_memory** and rewrite all tainted memory accesses into safe accesses at **dummy_memory**. This works for both reads and writes and it doesn't introduce side effects (as long as **dummy_memory** is not touched by the rest of the program). This may look as follows (**i** being the secret variable):

```
int safe_access(int* arr, int n, int i) {
  if (i < n)
    return arr[i];
  else
    return 0;
}
```

```
int safe_access(int* arr, int n, int i) {
  bool cond = (i < n);
  int* mem = cond ? arr : dummy_memory;
  int i_ = cond ? i : 0;
  int result_1 = mem[i_];
  int result_2 = 0;
  int result = cond ? result_1 : result_2;
  return result;
}
```

While this prevents crashes due to wrong memory accesses, it may still leave timing effects due to cache effects. Mitigating these is a hard problem with no good general solution available. There are solutions for common special cases (like for array accesses in a loop with static bounds), but the only general solution we can imagine is to just turn off the cache. We may investigate this further during AP4.

2.4.3 Function calls

A different problem arises when function calls are performed in a tainted context. A pure (side-effect free) function like **cos** could just be executed and the result be discarded if required, just as we do it for assignments. But because a function may, in general, execute side-effects, we cannot do this.

For internal function calls (where the compiler has access to the called function's code), we can mitigate this problem in two ways. The first way is to inline the full code of the function, which is of course impractical because the code size may grow arbitrarily. The second way is to generate a version of the function that takes an additional parameter **dummy**. This parameter is dynamically set to true exactly if the function is executed in a tainted context. Then, PCFL must also be applied to this function to ensure its execution time does not depend on **dummy**, and also to avoid side effects if **dummy** is true.

External function calls on the other hand, like **printf**, are problematic because we do not have access to their code and therefore cannot predicate them. We can decide to either leave those calls guarded by an if-statement – which results in the calling function not being timing-invariant – or to execute the function call and only predicate the assignment of the result – which may introduce unwanted side effects that change the semantics of the program. When we know that the function is side-effect free, we can of course safely go for the second option. Then, however, we still do not know whether the calling function is actually timing-invariant because the external function is not linearized, so it may take different time depending on the inputs.

[6] We could do a pointer analysis similar to what [7] is doing to detect valid accesses. This would definitely help in many cases, but it would not suffice for all possible cases as such an analysis can never be fully precise.

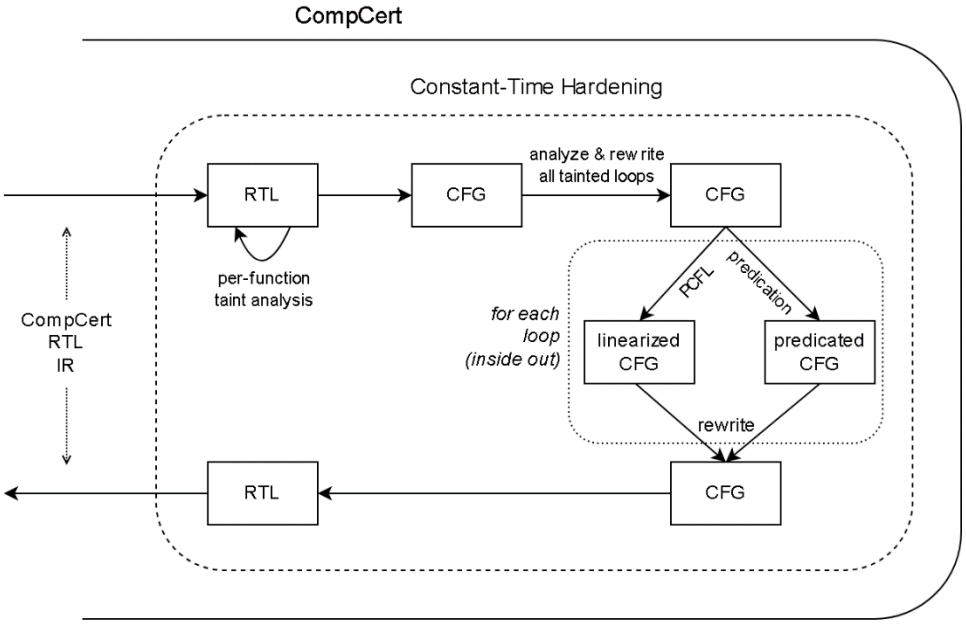
A good option is just to warn the code author if they use external functions. The code author could also instruct the compiler to “trust” certain external functions to be both pure and timing-invariant, e.g. through custom annotations.

2.5 Design in CompCert

We now briefly describe our CompCert implementation of the ideas and algorithms that have been described above, which is available as an artefact[7].

Our code mainly is written in OCaml, a language that interacts nicely with Coq. The main file, *TransformCT.ml*, contains the driver code of the transform. It provides the entry point *process_rtl_program* which is exposed to Coq and called from CompCert’s *Compiler.v*.

The transform works on the intermediate representation *RTL*. The following figure roughly sketches the algorithm that we implemented:



In more detail, our code performs the following steps:

1. Perform a rudimentary intra-function taint analysis to process the taint analysis results obtained from running Astrée as described in our report on AP2 [2]. This determines which variables and which if-conditions are tainted. (File *TaintAna.v*)
2. Convert the RTL code into a CFG, a data type that resembles a graph more closely. Detect and merge basic blocks. (File *CFG.ml*)
3. Prepare the CFG: remove empty blocks and do some loop transforms like making latch blocks unique. Then, analyse the loop structure of the CFG. (Files *LoopPrep.ml* and *LoopAna.ml*).
4. Clean all tainted loops by removing tainted loop exits and adding loop dummy-variables as described in section 2.3.2. (File *LoopPrep.ml*)
5. Calculate a loop- and dominance-compact block index of the CFG. Such a block index is a special case of a topological sort, which is required by PCFL. (Files *Dom.ml* and *PCFL.ml*)

[7] Available on the FZI-internal Git repository, tagged with “D3.1”

6. Perform PCFL using the CFG, the block index and the taint analysis results. This yields a transformed CFG. (File *PCFL.ml*)
7. Do the predication in a per-loop fashion: for each loop that was affected by the transform, first calculate a predication of the loop and then rewrite all instructions using the predicates, considering what was discussed in sections 2.3.1.2, 2.3.2 and 2.4. This includes various analyses over the CFG like liveness, definedness and a predicate simplification analysis to optimize the generated code. (Files *PredAna.ml*, *PredRewrite.ml* and *OpRewrite.ml*)
8. Convert the resulting CFG back into RTL code to be fed back into CompCert's compiler pipeline. (File *CFG.ml*).

For purposes of developing and debugging, we wrote tools that visualize the graph structure and allow comparing original and transformed graph (File *GraphViz.ml*). Additionally, we wrote a simulation that can simulate a function execution with different inputs and experimentally verify both that the transform doesn't change the semantics of the function and that the transformed function is actually timing-invariant (File *Simulation.ml*).

To minimise the risk that subsequent transforms destroy the timing-invariance that our transform has introduced, we perform the transform at the latest possible time during the RTL stage. We have not yet analysed the subsequent LTL, Mach and Asm transforms and their effect on timing-invariance – this could be done during AP4.

3 References

[1] FreeSBee Consortium, “D1.1 State of the Art: Timing and data flow side-channels,” 2023.

[2] FreeSBee-Konsortium, “D2.1 Concept for source-code transformation,” 2024.

[3] AbsInt Angewandte Informatik GmbH, “Fast and sound static analysis,” 2024. [Online]. Available: www.absint.com/astree.

[4] AbsInt Angewandte Informatik GmbH, “Formally verified compilation,” 2024. [Online]. Available: www.absint.com/compcert.

[5] S. Moll and S. Hack, “Partial Control-Flow Linearization,” in *PLDI*, Philadelphia, 2018.

[6] L. Soares, M. Canesche and F. M. Q. Pereira, “Side-channel Elimination via Partial Control-flow Linearization,” *ACM Transactions on Programming Languages and Systems,* vol. 45, pp. 1-43, 2023.

[7] L. Soares and F. M. Q. Pereira, “Memory-Safe Elimination of Side Channels,” in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Seoul, 2021.

[8] LLVM Project, “LLVM Loop Terminology (and Canonical Forms),” 2003-2024. [Online]. Available: https://llvm.org/docs/LoopTerminology.html. [Accessed 25 07 2024].