APPENDIX A
TYPE INFERENCE RULES

The security analysis of SecCG requires the type annotation of program variables and variables generated by the transformations of the underlying constraint-based compiler backend. We have implemented the type inference algorithm by Wang et al. [6] due to its scalability compared with other approaches like symbolic execution [5]. This section describes the type inference algorithm starting with the definition of auxiliary functions. Although SecCG uses multiple equivalent temporary (copy) values for each operation operand (see Figure 4), definitions use a single temporary value t. In reality, we unify these equivalent temporaries because they are semantically equal, as they are just copies of the original program variables. In the following definition, the parts in **bold** denote the extensions to the original type-inference algorithm [6].

The auxiliary function *xor*, returns true if an expression only consists of exclusive OR operations. This function improves the precision of the type inference algorithm, when multiple exclusive OR operations remove the dependence on a secret value. The recursive definition of *xor* is as follows:

$$\boldsymbol{xor}(t_0) = \begin{cases} \texttt{true} & \text{if } \boldsymbol{t_0 \in IN} \\ \boldsymbol{xor}(t_1) & \text{if } \boldsymbol{t_0 = uop(t_1)} \\ \boldsymbol{xor}(t_1) \wedge \boldsymbol{xor}(t_2) & \text{if } \boldsymbol{t_0 = \oplus(t_1, t_2)} \\ \texttt{false} & \text{if } \boldsymbol{t_0 = bop(t_1, t_2)}, \\ & \quad \boldsymbol{bop \neq \oplus} \end{cases}$$

The auxiliary function *supp* [6] returns the support of each expression. That is, all the variables that are syntactically present in the expression. We add two cases for *supp*, where some of syntactically present values are removed in the case of a simplification. This improves the precision of the analysis, because the type inference algorithm uses *supp* to decide on the type of a temporary variable. The recursive definition of *supp* is:

$$supp(t_0) = \begin{cases} \{t_0\} & \text{if } t_0 \in IN \\ supp(t_1) & \text{if } t_0 = uop(t_1) \\ \boldsymbol{(supp(t_1) \cup supp(t_2))\backslash} & \text{if } \boldsymbol{t_0 = \oplus(t_1, t_2) \wedge} \\ \boldsymbol{(supp(t_1) \cap supp(t_2))} & \quad \boldsymbol{xor(t_0)} \\ \boldsymbol{supp(t_2)} & \text{if } \boldsymbol{t_0 = \oplus(t_1, \oplus(t_1, t_2))} \\ supp(t_1) \cup supp(t_2) & \text{if } t_0 = bop(t_1, t_2) \end{cases}$$

The definitions of *unq* and *dom* are the same as the original definitions by Wang et al.. We define them here for completeness.

Auxiliary function *unq* [6] returns the random input variables that appear only once in the expression. This means that if we have a binary operator *bop*, with two operands t_1 and t_2 then, if both operands are randomized with the same random value, then this random value cannot randomize the expression t_0. The recursive definition of *unq* is:

$$unq(t_0) = \begin{cases} \{t_0\} & \text{if } t_0 \in IN_{rand} \\ \{\} & \text{if } t_0 \in IN \backslash IN_{rand} \\ unq(t_1) & \text{if } t_0 = uop(t_1) \\ (unq(t_1) \cup unq(t_2))\backslash & \\ (supp(t_1) \cap supp(t_2)) & \text{if } t_0 = bop(t_1, t_2) \end{cases}$$

The last auxiliary function is *dom* [6]. For each temporary variable, *dom* returns the random input variables that are xor:ed with that value. The recursive definition of *dom* is:

$$dom(t_0) = \begin{cases} \{t_0\} & \text{if } t_0 \in IN_{rand} \\ \{\} & \text{if } t_0 \in IN \backslash IN_{rand} \\ dom(t_1) & \text{if } t_0 = uop(t_1) \\ (dom(t_1) \cup dom(t_2)) & \\ \quad \cap\ unq(t_0) & \text{if } t_0 = \oplus(t_1, t_2) \\ \{\} & \text{if } t_0 = bop(t_1, t_2) \wedge bop \neq \oplus \end{cases}$$

Finally, Figure 10 presents the type system. Rules `RAND` and $\texttt{PUB}_1$ to $\texttt{PUB}_8$ are described by Wang et al. [6] and the rest of the rules are discussed in the same paper. Here, for space reasons, we have abbreviated `Random` to *Rand*, `Public` to *Pub*, and `Secret` to *Sec*. In particular, the first two rules are the basic rules, i.e. 1) if *dom* for an expression contains a value, then, this temporary has type *Rand*, and 2) if the type is not *Rand* and the expression does not depend on secret values, then the expression has type *Pub*. The rest of the rules improve the precision of the analysis. In particular, rules DISTR_0 to DISTR_3 are new rules that do not appear in Wang et al..

APPENDIX B
SECURITY PROOF

We assume that the type-inference algorithm [6] is conservative and sound: if $type(t) = Rand$, then t follows a uniform random distribution; if $type(t) = Pub$, then t follows a secret-independent distribution (might also be uniform random distribution); and if $type(t) = Sec$, then t may be secret dependent.

SecCG generates a solution to the constraint model, which we represent as an ordered sequence of instructions, $P = \{i_0, ..., i_n\}$. This means that instruction i_j is executed before instruction i_k for $j < k$.

To verify whether the generated program leaks secret information according to our leakage model (Equations 1-4), we give a proof of Theorem 3 using structural induction on a mitigated program, P. We start from the last instruction because preceding instructions are able to hide the secret values.

Case 1 Assume $P = t \leftarrow e$.
From the leakage model (Equation 3), we have $L(P(IN)) = \{HW(e \oplus r_{IN})\}$, where $r(t) = r$).

Case 1.a Assume $type(r_{IN}) = Pub$.
This means that the input is a constant value.

$$\frac{dom(t) \neq \varnothing}{\Gamma \vdash t : Rand}\ \text{RAND} \qquad \frac{supp(t) \cap IN_{sec} = \varnothing \quad dom(t) = \varnothing}{\Gamma \vdash t : Pub}\ \text{PUB}_1 \qquad \frac{\Gamma \vdash t_0 : Pub \quad \Gamma \vdash t_1 : Pub \quad supp(t_0) \cap supp(t_1) = \varnothing}{\Gamma \vdash t = t_0 \circledast t_1 : Pub}\ \text{PUB}_2$$

$$\frac{\Gamma \vdash t_0 : Rand \quad \Gamma \vdash t_1 : Rand \quad (dom(t_0) \backslash supp(t_1) \neq \varnothing \vee dom(t_1) \backslash supp(t_0) \neq \varnothing)}{\Gamma \vdash t_0 \circ t_1 : Pub}\ \text{PUB}_3 \qquad \frac{\Gamma \vdash t_0 : Rand \quad \Gamma \vdash t_1 : Rand \quad (dom(t_0) \backslash dom(t_1) \neq \varnothing \vee dom(t_1) \backslash dom(t_0) \neq \varnothing)}{\Gamma \vdash t_0 \odot t_1 : Pub}\ \text{PUB}_4 \qquad \frac{i \in \{0,1\} \quad j = 1 - i \quad \Gamma \vdash t_i : Rand \quad dom(t_i) \backslash supp(t_j) = \varnothing \quad dom(t_i) = dom(t_j) \quad supp(t_i) = supp(t_j)}{\Gamma \vdash t_0 \circledast t_1 : Pub}\ \text{PUB}_5$$

$$\frac{i \in \{0,1\} \quad j = 1 - i \quad \Gamma \vdash t_i : Rand \quad \Gamma \vdash t_j : Pub \quad dom(t_i) \backslash supp(t_j) \neq \varnothing}{\Gamma \vdash t_0 \odot t_1 : Pub}\ \text{PUB}_6 \qquad \frac{i \in \{0,1\} \quad j = 1 - i \quad \Gamma \vdash t_i : Pub \quad \Gamma \vdash t_j : Rand \quad supp(t_i) \cap supp(t_j) = \varnothing}{\Gamma \vdash t_0 \circ t_1 : Pub}\ \text{PUB}_7 \qquad \frac{(t_0 = t_1 \odot t_2 \vee t_1 = t_0 \odot t_2) \quad \Gamma \vdash t_{0,1} : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_1 \neq Sec \wedge T_2 \neq Sec}{\Gamma \vdash t_0 \oplus t_1 : Pub}\ \text{PUB}_8$$

$$\frac{((t_0 = t_1 \oplus t_2) \vee (t_1 = t_0 \oplus t_2)) \quad \Gamma \vdash t_2 : T}{\Gamma \vdash t_0 \oplus t_1 : T}\ \text{NEST}_1 \qquad \frac{((t_0 = t_1 \vee t_2) \vee (t_1 = t_0 \vee t_2)) \quad \Gamma \vdash \neg t_{0,1} \wedge t_2 : T}{\Gamma \vdash t_0 \oplus t_1 : T}\ \text{NEST}_2 \qquad \frac{(t_0 = t_1 \wedge t_2 \wedge t_1 = t_0 \wedge t_2) \quad \Gamma \vdash t_{0,1} \wedge \neg t_2 : T}{\Gamma \vdash t_0 \oplus t_1 : T}\ \text{NEST}_3$$

$$\frac{\Gamma \vdash t_0 \odot (t_1 \oplus t_2) : T}{\Gamma \vdash (t_0 \odot t_1) \oplus (t_0 \odot t_2) : T}\ \text{DISTR}_0 \qquad \frac{\Gamma \vdash t_0 \odot (t_1 \oplus t_2) : T}{\Gamma \vdash (t_0 \odot t_1) \oplus (t_2 \odot t_0) : T}\ \text{DISTR}_1 \qquad \frac{\Gamma \vdash t_1 \odot (t_0 \oplus t_2) : T}{\Gamma \vdash (t_0 \odot t_1) \oplus (t_1 \odot t_2) : T}\ \text{DISTR}_2$$

$$\frac{\Gamma \vdash t_1 \odot (t_0 \oplus t_2) : T}{\Gamma \vdash (t_0 \odot t_1) \oplus (t_2 \odot t_1) : T}\ \text{DISTR}_3 \qquad \frac{\Gamma \vdash t_0 : Pub \quad \Gamma \vdash t_1 : Pub \quad supp(t_0) \cap supp(t_1) \cap IN_{rand} = \varnothing}{\Gamma \vdash t = t_0 \oplus t_1 : Pub}\ \text{PUB}_9$$

Fig. 10: Type inference for power side channels in SecCG [6]; $\oplus$ denotes the exclusive OR operation, $\odot$ denotes the multiplication in a finite field, $\circ$ denotes any other operations apart from $\odot$ and $\oplus$, and finally, $\circledast$ denotes any operation.

Case 1.a.i Assume $type(e) \in \{Rand, Pub\}$.
Because $type(e) \in \{Rand, Pub\}$, the distribution of e is either random (uniformly distributed) or public. This means that the distribution is not dependent on the secret value. Thus, Definition 1 is satisfied.
Case 1.a.ii Assume $type(e) = Sec$.
From the definition of *Spairs* (Equation 8), $type(e) = type(t) = Sec \implies \exists (t_i, ts) \in Spairs.\ t_i = t \wedge ts = [t' | t' \in Temps \wedge type(t') = Rand \wedge type(t' \oplus t) = Rand]$. In this case, we have a pair (t, ts), but the first constraint in Section IV-B2 is not satisfied because $\nexists t_r \in ts\ .\ \texttt{subseq}(t_r, t)$ (Theorem 1). So, P is not a valid program.

Case 1.b Assume $type(r_{IN}) = Rand$.
Case 1.b.i Assume $type(e) \in \{Rand, Pub\}$.
Case 1.b.i.α Assume $type(e \oplus r_{IN}) \in \{Rand, Pub\}$.
Because $type(e \oplus r_{IN}) \in \{Rand, Pub\}$, the distribution is either random (uniformly distributed) or public. This means that the distribution is not dependent on the secret value. Thus, Definition 1 is satisfied.
Case 1.b.i.β Assume $type(e \oplus r_{IN}) = Sec$.
From the hypotheses in Case 1.b and Case 1.b.i and the definition of $Rpairs$, we have that $(t(r_{IN}), t) \in Rpairs$. This means that the constraint in Section IV-B1 is not satisfied because we have that $\texttt{subseq}(t(r_{IN}), t)$. Hence, P is not a valid program.
Case 1.b.ii Assume $type(e) = Sec$.
From the definition of *Spairs* (Equation 8), $type(e) = type(t) = Sec \implies \exists (t_i, ts) \in Spairs.\ t_i = t \wedge ts = [t' | t' \in Temps \wedge type(t') = Rand \wedge type(t' \oplus t) = Rand]$.
Case 1.b.ii.α Assume $type(e \oplus r_{IN}) \in \{Rand, Pub\}$.
Because $type(e \oplus r_{IN}) \in \{Rand, Pub\}$, the distribution is either random (uniformly distributed) or public. This means that the distribution is not dependent on the secret value. Thus, Definition 1 is satisfied.
Case 1.b.ii.β Assume $type(e \oplus r_{IN}) = Sec$.
From Theorem 1, we have that $\texttt{subseq}(t(r_{IN}), t)$. Also, there is no other $t' \in Temps$ such that $\texttt{subseq}(t', t)$, i.e. $\nexists t' \neq t(r_{IN})\ .\ \texttt{subseq}(t', t)$. From the first constraint in Section IV-B2, we have that $\exists t' \in ts\ .\ \texttt{subseq}(t', t)$. Which means that $t' = t(r_{IN})$. However, if $t(r_{IN}) \in ts$ then $type(e \oplus r_{IN}) = Rand$ (Equation 8), which is not true. Hence, P is not a valid program.

Case 1.c Assume $type(r_{IN}) = Sec$.
Case 1.c.i Assume $type(e \oplus r_{IN}) \in \{Rand, Pub\}$.
Because $type(e \oplus r_{IN}) \in \{Rand, Pub\}$, the distribution is either random (uniformly distributed) or public. This means that the distribution is not dependent on the secret value. Thus, Definition 1 is satisfied.
Case 1.c.ii Assume $type(e \oplus r_{IN}) = Sec$.
From the definition of *Spairs* (Equation 8), $type(r_{IN}) = type(t(r_{IN})) = Sec \implies \exists (t_i, ts) \in Spairs.\ t_i = t \wedge ts = [t' | t' \in Temps \wedge type(t') = Rand \wedge type(t' \oplus t) = Rand]$. From Theorem 1, we have that $\texttt{subseq}(t(r_{IN}), t)$. Also, there is no other $t' \in Temps$ such that $\texttt{subseq}(t(r_{IN}, t)$, i.e. $\nexists t' \neq t\ .\ \texttt{subseq}(t, t')$. From the second constraint in Section IV-B2, we have that $\exists t' \in ts\ .\ \texttt{subseq}(t(r_{IN}, t'))$. Which means that $t' = t$. However, if $t' \in ts$ then $type(t \oplus t(r_{IN})) = Rand$ (Equation 8), which is not

valid from hypothesis (Case 1.c.ii). Hence, P is not a valid program.

Case 2 Assume $P = mem(e_a, e)$.

Case 2.a Assume $type(e) \in \{Rand, Pub\}$ From the leakage model (Equation 4), we have $L(P(IN)) = \{HW(e)\}$. Because $type(e) \in \{Rand, Pub\}$, the distribution of e is either random (uniformly distributed) or public, i.e. a constant value. This means that the distribution is not dependent on the secret value. Thus, Definition 1 is satisfied.

Case 2.b Assume $type(e) = Sec$.

From the definition of *Mspairs* (Equation 10), $type(e) = Sec \implies \exists (o_i, os) \in Mspairs.\ tm(o_i) = e \wedge os = [o' | o' \in MemOperations \wedge type(tm(o')) = Rand \wedge type(tm(o') \oplus tm(o)) = Rand]$. In this case we have a pair $(o, \varnothing)$, and thus, the constraint in Section IV-B4 is not satisfied, because $\nexists o_i \in \varnothing$. So, P is not a valid program.

Case 3 Assume $P = P'; t \leftarrow e$.

Case 3.a Assume $type(e) = Sec$.

From the definition of *Spairs* (Equation 8), $type(e) = type(t) = Sec \implies \exists (t_i, ts) \in Spairs.\ t_i = t \wedge ts = [t' | t' \in Temps \wedge type(t') = Rand \wedge type(t' \oplus t) = Rand]$.

From the Spairs constraint in Section IV-B2, we have that $\exists t_r \in ts.\ l(t) \implies l(t_r) \wedge \texttt{subseq}(t_r, t)$. From Theorem 1, we have $\texttt{subseq}(t_r, t) \implies P = P''; t_r \leftarrow e_r; P'''; t \leftarrow e \wedge r(t) = r(t_r) \wedge \forall i \leftarrow P'''.\ i = t' \leftarrow e' \wedge r(t') \neq r(t)$. According to the leakage model (Equations 1), $L(P) = L(P''; t_r \leftarrow e_r; P''') \cup \{HW(t_r \oplus t)\}$. Because $t_r \in ts$, we have that $type(t_r \oplus t) = Rand$. This means that $t_r \oplus t$ has a uniform random distribution, and, thus, $HW(t_r \oplus t)$ does not leak. From the induction hypothesis, $\sum_{l \in L(P'(IN))} \mathbb{E}[l] = \sum_{l \in L(P'(IN'))} \mathbb{E}[l]$ and $\sum_{l \in L(P'(IN))} var[l] = \sum_{l \in L(P'(IN'))} var[l]$. Thus, $\sum_{l \in L(P(IN))} \mathbb{E}[l] = \sum_{l \in L(P(IN))} \mathbb{E}[l] + HW(t_r \oplus t) = \sum_{l \in L(P(IN'))} \mathbb{E}[l] + HW(t_r \oplus t) = \sum_{l \in L(P(IN'))} \mathbb{E}[l]$. Same is true for var. Thus, Definition 1 is satisfied.

Case 3.b Assume $type(e) \in \{Rand, Pub\}$.

Case 3.b.i Assume $\exists i \in P'.\ i = t' \leftarrow e' \wedge r(t) = r(t')$. Of the temporaries assigned to the same register, we select the temporary that is scheduled last before t, i.e. $P = P''; t_r \leftarrow e_r; P'''; t \leftarrow e \wedge \forall i \leftarrow P'''.\ i = t' \leftarrow e' \wedge r(t') \neq r(t)$

Case 3.b.i.α Assume $type(t \oplus t') \in \{Rand, Pub\}$.

In this case, the leakage model is $L(P) = L(P''; t_r \leftarrow e_r; P''') \cup \{HW(t \oplus t')\}$. Due to the initial assumption $type(t \oplus t') \in \{R, P\}$, the distribution of the leakage is either randomly distributed or public, i.e. it does not reveal secret information. From the induction hypothesis, $\sum_{l \in L(P'(IN))} \mathbb{E}[l] = \sum_{l \in L(P'(IN'))} \mathbb{E}[l]$ and $\sum_{l \in L(P'(IN))} var[l] = \sum_{l \in L(P'(IN'))} var[l]$. Thus, $\sum_{l \in L(P(IN))} \mathbb{E}[l] = \sum_{l \in L(P(IN))} \mathbb{E}[l] + HW(t \oplus t') = \sum_{l \in L(P(IN'))} \mathbb{E}[l] + HW(t \oplus t') = \sum_{l \in L(P(IN'))} \mathbb{E}[l]$. Same is true for var. Thus, Definition 1 is satisfied.

Case 3.b.i.β Assume $type(t \oplus t') = Sec$.

Case 3.b.i.β.1 Assume $type(t') \in \{Rand, Pub\}$.

From the definition of $Rpairs$ (Equation 7), $(t, t') \in Rpairs$. From the $Rpairs$ constraint in Section IV-B, we have that $\neg\texttt{subseq}(t, t') \wedge \neg\texttt{subseq}(t', t)$. From the definition of `subseq`, the first term, $\neg\texttt{subseq}(t, t')$, is `true` because t follows t' in the program sequence. The second constraint $\neg\texttt{subseq}(t', t)$ contradicts with the hypothesis in *Case 3.a.i* (Theorem 1).

Case 3.b.i.β.2 Assume $type(t') = Sec$.

From the definition of $Spairs$ (Equation 8) we have that $\exists (t_i, ts) \in Spairs.\ t_i = t'$ with $\forall t_s \in ts.\ type(t' \oplus t_s) = Rand$. From the $Spairs$ constraint in Section IV-B, $\exists t_r \in ts.\ l(t') \implies l(t_r) \wedge \texttt{subseq}(t', t_r)$. However, because there is no other assignment to register $r(t)$ in P''' (*Case 3.b.i*), we have that $t_r = t$ and because $t_r \in ts$, $type(t_r \oplus t') = Rand$. But $type(t \oplus t') = Sec$ (*Case 2.b.i*), which is a contradiction.

Case 3.b.ii Assume $\nexists i \in P'.\ i = t' \leftarrow e' \wedge r(t) = r(t')$. Then, the leakage is $L(P) = L(P') \cup \{HW(e)\}$. $HW(e)$ follows either a random distribution or is secret independent. From the induction hypothesis, $\sum_{l \in L(P'(IN))} \mathbb{E}[l] = \sum_{l \in L(P'(IN'))} \mathbb{E}[l]$ and $\sum_{l \in L(P'(IN))} var[l] = \sum_{l \in L(P'(IN'))} var[l]$. Thus, $\sum_{l \in L(P(IN))} \mathbb{E}[l] = \sum_{l \in L(P(IN))} \mathbb{E}[l] + HW(e) = \sum_{l \in L(P(IN'))} \mathbb{E}[l] + HW(e) = \sum_{l \in L(P(IN'))} \mathbb{E}[l]$. Same is true for var. Thus, Definition 1 is satisfied.

Case 4 Assume $P = P'; mem(e, e_i)$.

Case 4.a Assume $type(e) = Sec$.

Analogous to *Case 3.a*.

Case 4.b Assume $type(e) \in \{Rand, Pub\}$.

Analogous to *Case 3.b*.

Appendix C
Implied Constraints

To improve the solver's ability to find solutions, we add additional constraints that are logically implied by the imposed constraints. Implied constraints often improve the solving procedure by reducing the search space through propagation [22].

The following implied constraint is specifically relevant to ARM Cortex M0 but also to architectures that use accumulators for many operations, such as x86 architectures. This constraint enforces that if a pair of temporaries in $Rpairs$ belong to the same operation $\circ$ then the two operands (destination and source) have to be assigned to different registers or the operation operands should change. If the source and destination operands have to be assigned to the same register (accumulator) then, the operands have to be inverted. The constraint is as follows:

```
forall (t1,t2) in Rpairs:
  o = def_oper(t1)
  if (o ∈ user_opers(t2)):
    ¬same_reg(t1, t2)
```

Another implied constraint is related to preassigned operands. Preassigned operands are given a specific register because of special hardware architecture properties or calling conventions. For this, we add an additional implied constraint that guides the solver to try to schedule a different temporary if the two preassigned temporaries are not allowed to be subsequent, i.e. they belong to $Rpairs$.

```
forall (t1,t2) in Rpairs:
  if (t2 ∈ preassign ∧ t1 ∈ preassign):
    samereg(t1, t2) ⟹ (
     (exists t ∈ Temps: subseq(t1,t) ∨
                        subseq(t,t1)) ∧
     (exists t ∈ Temps: subseq(t2,t) ∨
                        subseq(t,t2)))
```