Quantifying and Preventing Side Channels with Substructural Type Systems

Van Chan Ngo   Mario Dehesa-Azuara   Matthew Fredrikson   Jan Hoffmann
Carnegie Mellon University

Abstract

Static techniques for deriving upper bounds on the resource consumption of programs have been extensively studied. However, there are applications that require more fine-grained information such as the difference between upper and lower bounds or the guaranty that the resource usage of a program does not differ for certain inputs. This article presents two novel substructural type systems for deriving lower bounds and for proving that a program has constant resource consumption for a class of inputs. The type systems are based on the potential method of amortized analysis to achieve compositionality, precision, and automatic inference using off-the-shelf linear optimization. While classic amortized analysis treats potential as an affine resource, the novel type systems treat potential as a relevant and linear resource, respectively. The soundness of the type systems with respect to an operational cost semantics is verified using the proof assistant Agda. The novel constant-resource and lower bound analyses are applied to quantify and prevent security vulnerabilities that leak secret information through resource consumption, such as side channels. First, implementations of the lower bound and constant-resource type systems in Resource Aware ML are used to automatically verify constant-time implementations of list comparison, encryption and decryption routines, database queries, and other resource-sensitive functionality. Second, the type systems are used to implement a method for automatically turning programs into constant-resource programs using LP solving. The method is static, does not require tracking resources at runtime, and works on most programs for which Resource Aware ML can derive an upper bound. Third, a resource-aware noninterference property is introduced. It relaxes the constant-resource requirement on programs, and requires only that resource usage does not leak information about *secret* inputs. This property is statically verified by combining the linear type system for constant resource consumption with an information flow type system.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal Methods; D.4.6 [*Security and protection*]: Information Flow Controls

Keywords Timing channels, resource analysis, information flow

1. Introduction

Automatic static analysis of the resource consumption of programs is an active area of research. Motivated by applications in embedded and real-time systems [85], finding performance bugs [69], and providing feedback to developers [41], static resource analysis techniques have focused on derivation of worst-case bounds [2, 4, 13, 14, 20, 27, 33, 77]. One successful technique for automatically finding resource bounds at compile time is automatic amortized resource analysis (AARA). The idea of AARA is to combine the potential method of amortized analysis with existing programming languages techniques to achieve automation. For example, AARA has been integrated into type systems to automatically derive linear [51] and polynomial [47, 48, 50] bounds for strict and higher-order [9, 57] functional programs.

The main advantages of AARA are compositionality, efficiency, and precision. It has been shown that the technique can automatically derive bounds for complex real-world programs such as parts of the CompCert C Compiler [9] and the cBench benchmark suite [26]. Precision and efficiency stems from the selection of algebraic structures such as multivariate resource polynomials [50] that can represent a wide range of bounds, as well as the reduction of bound inference to efficient LP solving. AARA is naturally compositional since the potential methods integrates reasoning about size changes and resource consumption. However, *existing AARA techniques are limited to worst-case bounds.*

Novel resource type systems The starting point of this paper is the technical insight that *the potential method of amortized analysis can also be used to derive lower bounds*, as well as to *prove that a program has constant resource consumption for a fixed input size.* In classic AARA the potential is used as an affine resource: it must be available to cover cost but excess potential is simply discarded. We show that if potential is treated as a linear resource, then corresponding type derivations prove that programs have *constant* resource consumption, i.e., resource consumption is independent of the execution path. Intuitively, this amounts to requiring that *all* potential must be used to cover the cost and that excess potential is not wasted. Furthermore, we show that if potential is treated as a relevant resource, then we derive *lower bounds* on the resource usage. Following a similar intuition, this requires that all potential is used, but the available potential does not need to be sufficient to cover the remaining cost.

The two novel type systems that we present enjoy the same advantages as classic AARA for upper bounds. They are naturally compositional, often derive precise results, and allow for fully-automated type inference based on LP solving. Moreover, as in classic AARA, they are parametric in the resource of interest and incorporate user-specified resource metrics that assign a constant cost to each basic operation. The type systems discussed in this paper apply to a simple first-order functional language, and use the linear potential annotations from the original work of Hofmann and Jost [51]. This is sufficient to discuss the main technical points although it limits the systems to linear bounds. However, our implementation builds on Resource Aware ML (RAML) [9], and supports polynomial bounds, user-defined data types, and higher-order functions. We formalized the soundness proof of these type systems, as well as that of classic linear AARA, in the proof assistant Agda. Soundness is proved with respect to an operational cost semantics, and like the type systems themselves, is parametric in the resource of interest.

Side channel mitigation In the second half of the paper, we apply our lower-bound and constant-resource type systems to the problem of preventing and quantifying *side channel vulnerabilities.* Side channel attacks extract sensitive information about a program's state through its use of resources such as time, network, and memory. Several notable instances of this type of attack have demonstrated leakage of cryptographic keys [5, 21, 24, 40, 58] and private user data [8, 19, 35, 42, 89] through such channels.

Whereas traditional notions of information flow can be described in terms of standard program semantics, a similar treatment of side channels requires incorporating the corresponding resource into the semantics and applying quantitative reasoning. This difficulty has led previous work in the area to treat resource use indirectly, by reasoning about the flow of secret information into branching control flow [6, 16, 66, 73] or introducing obfuscation components that mask secret-dependent differences in resource use [11, 61]. These approaches can limit program expressiveness or lead to unnecessary performance penalties.

In contrast, our approach performs quantitative analysis of resource use directly through the constant-resource type system. We consider an adversary that is able to observe the final resource consumption of a program as specified by a cost semantics, and derive a proof that the attacker's observations will not change as the program's inputs do. Although this observation model does not cover all known side-channel attacks, it applies to a large class of attackers that are not able to make intermediate observations of the program's behavior, such as those that reside over a network. Additionally, we show how one can use derived upper and lower bounds to quantify leakage through resource use, by reasoning about the number of distinct observations an attacker can make.

In general, requiring that a program only ever consumes a constant amount of resources is too restrictive. In most settings, it is sufficient to make sure that the resource usage of a program does not depend on selected parts of the input. To account for this, we present a new information flow type system that incorporates our constant-resource type system to reason about an adversary who can observe and manipulate inputs marked public, but can only make observations on secret inputs through the program's resource behavior and public outputs. Intuitively, the guarantee enforced by this type system, *resource-aware noninterference*, requires that the parts of the program affected by secret inputs can only make constant use of resources.

The main technical contribution in this part is the soundness proof of this type system with respect to the cost semantics. The main conceptional contribution is that the type system allows to freely switch between local and global reasoning. One extreme would be to ignore the information flow of the secret values and prove that the whole program has constant resource consumption. The other extreme would be to ensure that every conditional that branches on a secret value (a critical conditional) uses a constant amount of resources. However, there are constant time programs in which individual conditional are not constant time (see Section 4). As a result, we allow different levels of global and local reasoning and in the type system to ensure that every critical conditional occurs in a constant-resource block.

Finally, we show that our type inference algorithm for the constant-resource type system can be used to automatically turn programs into constant-resource programs. To this end, we introduce a *consume* expression that performs resource padding (e.g., sleep for time). The amount of resource padding that is needed is automatically determined by the LP solver and is parametric in the size of the program variables. This technique is more efficient then existing techniques [citations] since it does not change the worst-case resource behavior of the program. Of course, it would be possible to do such a resource padding to the worst-case behavior dynamically at the end of the run of the program. The advantage of our method is that we do not have to keep track of the actual resource usage at runtime and that we automatically derive a proof (a type derivation) that the modified program has constant resource use without reasoning on a meta level. We implemented this technique in RAML.

Contributions We make the following contributions:

- Two novel AARA type systems that derive lower bounds and prove constant resource use, and an implementation of these systems that extends RAML. We evaluate the implementation on several examples, including encryption routines and data processing programs that were previously studied in the context of timing leaks in differentially-private systems [42].
- A mechanization of the soundness proofs the two new type systems and classic AARA for upper bounds in Agda. To the best our knowledge, this is also the first formalization of the soundness of linear AARA for worst-case bounds.
- An information-flow type system that incorporates our constant-time system to prevent leakage of selected secrets through resource side channels, and an LP-based method that transforms programs into constant-resource versions.

Technical details including the complete proofs and inference rules can found in a companion technical report [10] .

2. Language-level constant-resource programs

In this section we introduce a language, an operational cost semantics, and the notion of constant-resource functions.

2.1 The language

Syntax To discuss the main ideas of our work, it is sufficient to study a purely functional first-order and monomorphic typed functional language with Booleans, integers, pairs, and list data types, pattern matching and recursive functions as given in Fig. 1.

$$
\begin{array}{lll}
T & ::= & \mathrm{unit} \mid \mathrm{bool} \mid \mathrm{int} \mid L(T) \mid T * T \\
G & ::= & T \to T \\
e & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid x \mid \mathrm{op}_\diamond(x_1, x_2) \mid \mathrm{app}(f, x) \mid \mathrm{if}(x, e_t, e_f) \\
 & & \mid \mathrm{let}(x, e_1, x.e_2) \mid \mathrm{pair}(x_1, x_2) \mid \mathrm{match}(x, (x_1, x_2).e) \mid \mathrm{nil} \\
 & & \mid \mathrm{cons}(x_1, x_2) \mid \mathrm{match}(x, e_1, (x_1, x_2).e_2) \mid \mathrm{share}(x, (x_1, x_2).e) \\
v & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid \mathsf{nil} \mid [v_1, \ldots, v_n] \mid (v_1, v_2) \\
\diamond & \in & \{+, -, *, \mathsf{div}, \mathsf{mod}, =, <>, >, <, <=, >=, \mathsf{and}, \mathsf{or}\}
\end{array}
$$

Figure 1. Syntax of the language

In this grammar we use abstract binding trees [43] and in examples we use equivalent expressions in OCaml syntax. The expressions are in *let normal form*, meaning that they are formed from variables whenever it is possible. It makes the typing rules and semantics simpler without loosing expressivity. The syntactic form *share* has to be use to introduce multiple occurrences of a variable in an expression.

A value is a boolean constant, an integer value n, the empty list nil, a list of values $[v_1, \ldots, v_n]$, or a pair of values (v_1, v_2). A type context $\Gamma : \mathrm{VID} \to \mathscr{T}$ is a partial mapping from variable identifiers to data types T. A *signature* $\Sigma : \mathrm{FID} \to \mathscr{G}$ is a partial mapping from function identifiers to first-order types G. The typing rules that define a type judgement $\Sigma; \Gamma \vdash e : T$ are standard.

A program is a tuple containing a signature Σ and a finite set of tuples $(e_g, x^g)_{g \in \mathrm{dom}(\Sigma)}$ where e_g is an expression defining the function's body and x^g is the argument. For any e_g, it holds that $\Sigma; x^g : T_1 \vdash e_g : T_2$ if $\Sigma(g) = T_1 \to T_2$.

Operational cost semantics The operational cost semantics defines the resource consumption of programs. It is instrumented with a non-negative resource counter that is incremented or decremented by a constant at every step of the semantics. The semantics is parametric in the cost that is used at each step and we call a particular set of such cost parameters a *cost model*. The constants can be used to indicate the costs of storing or loading a value in the memory, evaluating a primitive operation, binding of a value in the environment, or branching on a Boolean value.

(E:Bin)
$$\frac{v = E(x_1) \diamond E(x_2)}{E \left|\frac{q+K^{\mathsf{op}}}{q}\right. \mathrm{op}_{\diamond}(x_1, x_2) \Downarrow v}$$

(E:Fun)
$$\frac{\Sigma(g) = T_1 \to T_2 \qquad [y^g \mapsto E(x)] \left|\frac{q}{q'}\right. e_g \Downarrow v}{E \left|\frac{q+K^{\mathsf{app}}}{q'}\right. \mathrm{app}(g, x) \Downarrow v}$$

(E:Let)
$$\frac{E \left|\frac{q-K^{\mathsf{let}}}{q'_1}\right. e_1 \Downarrow v_1 \qquad E[x \mapsto v_1] \left|\frac{q'_1}{q'}\right. e_2 \Downarrow v}{E \left|\frac{q}{q'}\right. \mathrm{let}(x, e_1, x.e_2) \Downarrow v}$$

(E:Var)
$$\frac{x \in \mathrm{dom}(E)}{E \left|\frac{q+K^{\mathsf{var}}}{q}\right. x \Downarrow E(x)}$$

(E:If-True)
$$\frac{E(x) = \mathsf{true} \qquad E \left|\frac{q-K^{\mathsf{cond}}}{q'}\right. e_t \Downarrow v}{E \left|\frac{q}{q'}\right. \mathrm{if}(x, e_t, e_f) \Downarrow v}$$

(E:Match-L)
$$\frac{E(x) = [v_1, ..., v_n] \qquad E[x_h \mapsto v_1, x_t \mapsto [v_2, ..., v_n]] \left|\frac{q-K^{\mathsf{matchL}}}{q'}\right. e_2 \Downarrow v}{E \left|\frac{q}{q'}\right. \mathrm{match}(x, e_1, (x_h, x_t).e_2) \Downarrow v}$$

Figure 2. Evaluation rules: operational cost semantics

It is possible to further parameterize some constants to obtain a more precise cost model. For example, the cost of calling a function may vary according to the number of the arguments. In the following, we will show that any suitable values can be used for the constants in the cost model and the soundness of the type system does not rely on any specific values for these constants. In the examples, we use a cost model in which the constants are either 1 for each step or 0 for all steps except for calls to the *tick* where *tick(q)* means that we have resource resource usage $q \in \mathbb{Q}$.

The cost semantics is based on a big-step semantics and is formulated using an *environment* $E : \mathrm{VID} \to \mathit{Val}$ that is a finite mapping from a set of variable identifiers to a set of values. Evaluation judgements are of the form $E \left|\frac{q}{q'}\right. e \Downarrow v$ where $q, q' \in \mathbb{Q}_0^+$. The intuitive meaning is that under the environment E and q available resources, e evaluates to the value v without running out of resources and q' resources are available after the evaluation. The evaluation consumes $\delta = q - q'$ resource units. Fig. 2 presents some selected evaluation rules. All rules are given in the TR [10].

2.2 Constant-resource programs

Informally, a program is *constant resource* if it has the same quantitative resource consumption under all environments in which values have the same size.

We write $\models v : T$ to denote that v is a well-formed value of type T. The typing rules for values are standard [10, 49, 51] and we omit them here. Let Γ be a context that maps variable identifiers to base types, an environment E is *well-formed* w.r.t Γ, denoted $\models E : \Gamma$, if $\forall x \in \mathrm{dom}(\Gamma).\ \models E(x) : \Gamma(x)$. Below we define the notation of *size equivalence*, written $|v| \approx |u|$, which is a binary relation relating two values v and u of the same type T.

$$\frac{T \in \{\mathrm{unit}, \mathrm{bool}, \mathrm{int}\}}{|v| \approx |u|} \qquad \frac{|v_1| \approx |u_1| \qquad |v_2| \approx |u_2|}{|(v_1, v_2)| \approx |(u_1, u_2)|}$$

$$\frac{m = n \qquad |v_i| \approx |u_i|}{|[v_1, ... v_n]| \approx |[u_1, ... u_m]|}$$

Let $X \subseteq \mathrm{dom}(\Gamma)$ be a set of variables and E_1, E_2 be two well-formed environments. Then E_1 and E_2 are *size-equivalent* w.r.t X, denoted $E_1 \approx_X E_2$, when they agree on the sizes of the variables in X, that is, $\forall x \in X. |E_1(x)| \approx |E_2(x)|$.

Definition 1. *An expression e is constant resource w.r.t $X \subseteq$ $\mathit{dom}(\Gamma)$, written $\mathit{const}_X(e)$, if for all well-formed environments E_1 and E_2 such that $E_1 \approx_X E_2$, the following statement holds.*

$$\textit{If } E_1 \left|\frac{p_1}{p'_1}\right. e \Downarrow v_1 \textit{ and } E_2 \left|\frac{p_2}{p'_2}\right. e \Downarrow v_2 \textit{ then } p_1 - p'_1 = p_2 - p'_2$$

We say that a function $g(x_1, \ldots, x_n) = e_g$ is constant resource w.r.t $X \subseteq \{x_1, \ldots, x_n\}$ if $\mathrm{const}_X(e_g)$. If $Y \subseteq X$ and $E_1 \approx_X E_2$ then $E_1 \approx_Y E_2$. Thus we have the following lemma.

Lemma 1. *For all e, X, and $Y \subseteq X$, if $\mathit{const}_Y(e)$ then $\mathit{const}_X(e)$.*

```
let rec compare(h,l) = match h with
  | [] → (match l with | [] → Raml.tick 1.0; true
    | y::ys → Raml.tick 1.0; false)
  | x::xs → match l with | [] → Raml.tick 1.0; false
    | y::ys → if (x = y) then
        Raml.tick 5.0; compare(xs,ys)
      else Raml.tick 5.0; false

let rec p_compare(h,l) =
 let rec aux(r,h,l) = match h with
  | [] → (match l with | [] → Raml.tick 1.0; r
    | y::ys → Raml.tick 1.0; false)
  | x::xs → match l with | [] → Raml.tick 1.0; false
    | y::ys → if (x = y) then
        Raml.tick 5.0; aux(r,xs,ys)
      else Raml.tick 5.0; aux(false,xs,ys)
  in aux(true,h,l)
```

Figure 3. The list comparison function *compare* is not constant resource, while the manually padded function *p_compare* is constant resource w.r.t h and l.

Example. *The function compare in Fig. 3 is not constant-resource function w.r.t h and l when the cost model is defined using tick annotations. Since the execution cost of the two branches of the conditional dependents on the relation of x and y. The function p_compare is a manually padded version with a dummy computation that is constant w.r.t h and l. However, it is not constant w.r.t h. For instance, p_compare([1;2;3],[0;1;2]) has cost* 16 *but p_compare([1;2;1],[0;1]) has cost* 12 $\neq$ 16. *If we further pad the nil case with Raml.tick 5.0; aux false xs [] to make the function to always iterate all of h's nodes, then it is constant w.r.t h.*

In Section 5, we will provide a better way to transform a program into constant with our extended expression *consume*. Users insert *consume* expressions into program-under-consideration then our analyzer will infer automatically the amount of resource units needed to spend to make the program constant.

3. Type systems for lower bounds and constant resource usage

In this section we introduce two substructural resource-annotated type systems: The type system for constant resource usage is linear and the one for lower bounds is relevant.

3.1 Background

Amortized analysis The potential method of amortized analysis has been introduced [79] to bound the *worst-case resource usage* of a sequence of data structure operations. The key idea is to incorporate a non-negative potential into the analysis that can be used to pay (costly) operations.

```
let rec filter_succ l = match l with
  | [] → Raml.tick 1.0; []
  | x::xs →
    if x > 0 then Raml.tick 8.0; filter_succ xs
    else Raml.tick 3.0; (x+1)::filter_succ xs

let fs_twice l = filter_succ (filter_succ l)
```

Figure 4. Two OCaml functions with linear resource usage. The *worst-case* number of *ticks* executed by *fitler_succ(ℓ)* and *fs_twice(ℓ)* is $8|\ell|+1$ and $11|\ell|+2$ respectively. In the *best-case* the functions execute $3|\ell|+1$ and $6|\ell|+2$ ticks, respectively. The resource consumption is not constant.

To statically analyze a program with the potential method, a mapping from program points to potentials must be established. One has to show that the potential at every program point suffices to cover the cost of any possible evaluation step and the potential of the next program point. The initial potential is then an upper bound on the resource usage of the program.

Linear potential for upper bounds To automate amortized analysis, we fix a format of the potential functions and use LP solving to find the optimal coefficients. To infer linear potential functions, inductive data types are annotated with a non-negative rational numbers q [51]. For example, the type $L^q(\text{bool})$ of Boolean lists with potential q defines potential $\Phi([b_1,\ldots,b_n] : L^q(\text{bool})) = q\cdot n$.

This idea is best explained by example. Consider the function *filter_succ* below that filters out positive numbers and increments non-positive numbers. As in RAML, we use OCaml syntax and *tick* commands to specify resource usage. If we filter out a number then we have a high cost (8 resource units) since *x* is, e.g., sent to an external device. If *x* is incremented we have a lower cost of 3 resource units. As a result, the worst-case resource consumption of *filter_succ(ℓ)* is $8|\ell|+1$ (where 1 is for the cost that occurs in the nil case of the match). The function *fs_twice(ℓ)* applies *filter_succ* twice, to ℓ and to the result of *filter_succ(ℓ)*. The worst-case behavior appears if no list element is filtered out in the first call and all elements are filtered out in the second call. The worst-case behavior is thus $11|\ell|+2$. These upper bounds can be expressed with the following annotated function types, which can be derived using local type rules in Fig. 5.

$$\begin{aligned}\text{filter\_succ}: &\quad L^8(\text{int}) \xrightarrow{1/0} L^0(\text{int})\\ \text{fs\_twice}: &\quad L^{11}(\text{int}) \xrightarrow{2/0} L^0(\text{int})\end{aligned}$$

Intuitively, the first function type states that an initial potential of $8|\ell|+1$ is sufficient to cover the cost of *filter_succ(ℓ)* and there is $0|\ell'|+0$ potential left where ℓ' is the result of the computation. This is just one possible potential annotation of many. The right choice of the potential annotation depends on the use of the function result. For example, for the inner call of *filter_succ* in *fs_twice* we need the following annotation.

$$\text{filter\_succ}: \quad L^{11}(\text{int}) \xrightarrow{2/1} L^8(\text{int})$$

It states that the initial potential of $11|\ell|+2$ is sufficient to cover the cost of *filter_succ(ℓ)* and there is $8|\ell'|+1$ potential left to be assigned to the returned list ℓ'. The potential of the result can then be used with the previous type of *filter_succ* to pay for the cost of the outer call.

$$\text{filter\_succ}: \quad L^p(\text{int}) \xrightarrow{q/q'} L^r(\text{int}) \mid q \geq q'+1 \wedge p \geq 8 \wedge p \geq 3+r$$

We can summarize all possible types of *filter_succ* with a linear constraint system. In the type inference, we generate such a constraint system and solve it with an off-the-shelf LP solver to derive a concrete bound. To obtain tight bounds, we perform a whole-program analysis and minimize the coefficients in the input potential.

Surprisingly, this approach—as well as the new concepts we introduce here—can be extended to polynomial bounds [50], higher-order functions [9, 57], polymorphism [56], and user-defined inductive types [9, 56].

3.2 Resource annotations

The resource-annotated types are base types in which the inductive data types are annotated with non-negative rational numbers, called *resource annotations*.

$$A ::= \text{unit} \mid \text{bool} \mid \text{int} \mid L^p(A) \mid A * A \quad (\text{for } p \in \mathbb{Q}_0^+)$$

A type context, $\Gamma^r : \text{VID} \to \mathscr{A}$, is a partial mapping from variable identifiers to resource-annotated types. The *underlying* base type and base type context denoted by $\widehat{A}$, and $\widehat{\Gamma^r}$ respectively can be obtained by removing the annotations. We extend all definitions such as $|v|$, $\models E : \Gamma$ and $\approx$ for base data types to resource-annotated data types by ignoring the annotations.

We now formally define the notation of *potential* representing how resource is associated with runtime values. The potential of a value v of type A, written $\Phi(v : A)$, is defined by the function $\Phi : \mathit{Val} \to \mathbb{Q}_0^+$ as follows.

$$\begin{aligned}&\Phi(() : \text{unit}) = \Phi(b : \text{bool}) = \Phi(n : \text{int}) = 0\\ &\Phi((v_1, v_2) : A_1 * A_2) = \Phi(v_1 : A_1) + \Phi(v_2 : A_2)\\ &\Phi([v_1, \cdots, v_n] : L^p(A)) = n\cdot p + \Sigma_{i=1}^n \Phi(v_i : A)\end{aligned}$$

Example. *The potential of a list $v = [b_1,\cdots,b_n]$ of type $L^p(bool)$ is $n\cdot p$. Similarly, a list of lists of Booleans values $v = [v_1,\cdots,v_n]$ of type $L^p(L^q(bool))$, where $v_i = [b_{i1},\cdots,b_{im_i}]$, has the potential $n\cdot p + (m_1 + \cdots + m_n)\cdot q$.*

Let Γ^r be a context and E be a well-formed environment w.r.t Γ^r. The potential of $X \subseteq \text{dom}(\Gamma^r)$ under E is defined as $\Phi_E(X : \Gamma^r) = \Sigma_{x\in X}\Phi(E(x) : \Gamma^r(x))$. The potential of Γ^r is $\Phi_E(\Gamma^r) = \Phi_E(\text{dom}(\Gamma^r) : \Gamma^r)$. Note that if $x \notin X$ then $\Phi_E(X)) = \Phi_{E[x\mapsto v]}(X)$. The following lemma states that the potential is the same under two well-formed size-equivalent environments.

Lemma 2. *If $E_1 \approx_X E_2$ then $\Phi_{E_1}(X : \Gamma^r) = \Phi_{E_2}(X : \Gamma^r)$.*

Annotated first-order data types are given as follows, where q and q' are rational numbers.

$$F ::= A_1 \xrightarrow{q/q'} A_2$$

A resource-annotated signature $\Sigma^r : \text{FID} \to \wp(\mathscr{F}) \setminus \{\emptyset\}$ is a partial mapping from function identifiers to a non-empty sets of annotated first-order types. That means a function can have different resource annotations depending on the context. The *underlying* base types are denoted by $\widehat{F}$. and the underlying base signature is denoted by $\widehat{\Sigma^r}$ where $\widehat{\Sigma^r}(f) = \widehat{\Sigma^r(f)}$.

3.3 Type system for constant resource consumption

The typing rules of the constant-resource type system define judgments of the form

$$\Sigma^r; \Gamma^r \,\Big|\frac{q}{q'}\; e : A$$

where e is an expression and $q, q' \in \mathbb{Q}_0^+$. The intended meaning is that in the environment E, $q + \Phi_E(\Gamma^r)$ resource units are sufficient to evaluate e to a value v with type A and there are *exactly* $q' + \Phi(v : A)$ resource units left over.

The typing rules form a *linear* type system. It ensures that every variable is used exactly once by allowing exchange but not weakening or contraction [84]. The rules can be organized into syntax directed and structural rules.

Syntax-directed rules The syntax-directed rules are shared among all type systems and selected rules are listed in Fig. 5.

$$\text{(A:VAR)}\quad \frac{}{\Sigma^r; x : A \vdash^{K^{\mathsf{var}}}_{0} x : A}
\qquad
\text{(A:B-OP)}\quad \frac{\diamond \in \{\mathsf{and}, \mathsf{or}\}}{\Sigma^r; x_1 : \text{bool}, x_2 : \text{bool} \vdash^{K^{\mathsf{op}}}_{0} \text{op}_\diamond(x_1, x_2) : \text{bool}}
\qquad
\text{(A:CONS)}\quad \frac{}{\Sigma^r; x_h : A, x_t : L^p(A) \vdash^{p+K^{\mathsf{cons}}}_{0} \text{cons}(x_h, x_t) : L^p(A)}$$

$$\text{(A:LET)}\quad \frac{\Sigma^r; \Gamma^r_1 \vdash^{q-K^{\mathsf{let}}}_{q'_1} e_1 : A_1 \qquad \Sigma^r; \Gamma^r_2, x : A_1 \vdash^{q'_1}_{q'} e_2 : A_2}{\Sigma^r; \Gamma^r_1, \Gamma^r_2 \vdash^{q}_{q'} \text{let}(x, e_1, x.e_2) : A_2}
\qquad
\text{(A:FUN)}\quad \frac{\Sigma^r(f) = A_1 \xrightarrow{q/q'} A_2}{\Sigma^r; x : A_1 \vdash^{q+K^{\mathsf{app}}}_{q'} \text{app}(f, x) : A_2}$$

$$\text{(A:IF)}\quad \frac{\Sigma^r; \Gamma^r \vdash^{q-K^{\mathsf{cond}}}_{q'} e_t : A \qquad \Sigma^r; \Gamma^r \vdash^{q-K^{\mathsf{cond}}}_{q'} e_f : A}{\Sigma^r; \Gamma^r, x : \text{bool} \vdash^{q}_{q'} \text{if}(x, e_t, e_f) : A}$$

$$\text{(A:MATCH-L)}\quad \frac{\Sigma^r; \Gamma^r \vdash^{q-K^{\mathsf{matchN}}}_{q'} e_1 : A_1 \qquad \Sigma^r; \Gamma^r, x_h : A, x_t : L^p(A) \vdash^{q+p-K^{\mathsf{matchL}}}_{q'} e_2 : A_1}{\Sigma^r; \Gamma^r, x : L^p(A) \vdash^{q}_{q'} \text{match}(x, e_1, (x_h, x_t).e_2) : A_1}
\qquad
\text{(A:SHARE)}\quad \frac{\Sigma^r; \Gamma^r, x_1 : A_1, x_2 : A_2 \vdash^{q}_{q'} e : B \qquad \curlyvee(A \mid A_1, A_2)}{\Sigma^r; \Gamma^r, x : A \vdash^{q}_{q'} \text{share}(x, (x_1, x_2).e) : B}$$

Figure 5. Selected syntax-directed rules of the resource type systems.

Rules like A:VAR and A:B-OP for leaf expressions (e.g., variable, binary operations, pairs) have fixed costs as specified by the constants K^x. Note that we require all available potential to be spent. The cost of the function call is represented by the constant K^{app} in the rule A:FUN and the argument carries the potential to pay for the function execution. In the rule A:LET, the cost of binding is represented by the constant K^{let}. The potentials carried by the contexts Γ^r_1 and Γ^r_2 are passed sequentially through the sub derivations. Note that the contexts are disjoint since our type system is linear. Multiple uses of variables must be introduced through the rule A:SHARE. The rule A:IF is the key rule for ensuring constant resource usage. By using the same context Γ^r for typing both e_t and e_f, we ensure that the conditional expression has the same resource usage in size-equivalent environments independent of the value of the Boolean variable x. The rules for inductive data types are crucial for the interaction of the linear potential annotations with the constant potential. The rule A:CONS shows how constant potential can be associated with a new data structure. The dual is the rule A:MATCH-L, which shows how potential associated with data can be released. It is important that these transitions are made in a linear fashion: potential is neither lost or gained.

Sharing The *share expression* makes multiple uses of a variable explicit. While multiple uses of a variable seem to be in conflict with the linear type discipline, the *sharing relation* $\curlyvee(A \mid A_1, A_2)$ ensures that potential is treated in a linear way. It apportions potential to ensure that the total potential associated with all uses is equal to the potential initially associated with the variable. This relation is only defined for structurally-identical types which differ in at most the resource annotations as follows.

$$\frac{A \in \{\text{unit}, \text{bool}, \text{int}\}}{\curlyvee(A \mid A, A)}
\qquad
\frac{\curlyvee(A \mid A_1, A_2) \qquad p = p_1 + p_2}{\curlyvee(L^p(A) \mid L^{p_1}(A_1), L^{p_2}(A_2))}$$

$$\frac{\curlyvee(A \mid A_1, A_2) \qquad \curlyvee(B \mid B_1, B_2)}{\curlyvee(A * B \mid A_1 * B_1, A_2 * B_2)}$$

Structural rules To allow more programs to be typed we add two structural rules to the type system which can be applied to every expression. These rules are specific to the the constant-resource type system.

$$\text{(C:WEAKENING)}\quad \frac{\Sigma^r; \Gamma^r \vdash^{q}_{q'} e : B \qquad \curlyvee(A \mid A, A)}{\Sigma^r; \Gamma^r, x : A \vdash^{q}_{q'} e : B}
\qquad
\text{(C:RELAX)}\quad \frac{\Sigma^r; \Gamma^r \vdash^{p}_{p'} e : A \qquad q \geq p \qquad q - p = q' - p'}{\Sigma^r; \Gamma^r \vdash^{q}_{q'} e : A}$$

The rule C:RELAX reflects the fact that if it is sufficient to evaluate e with p available resource units and there are p' resource units left over then e can be evaluated with $p + c$ resource units and there are exactly $p' + c$ resource left over, where $c \in \mathbb{Q}^+_0$. Rule C:WEAKENING states that an extra variable can be added into the given context if its potential is zero. The condition is enforced by $\curlyvee(A \mid A, A)$ since $\Phi(v : A) = \Phi(v : A) + \Phi(v : A)$ or $\Phi(v : A) = 0$. The rules can be used in branchings such as the conditional or the pattern match to ensure that subexpressions are typed using the same contexts and potential annotations.

Example. *Consider again the function p_compare in Fig. 3 in which the resource consumption is defined using* tick *annotations. The resource usage of p_compare(h, ℓ) is constant w.r.t h, that is, it is exactly* $5|h| + 1$. *This can be reflected by the following type.*

$$\mathsf{p\_compare}: \quad (L^5(int), L^0(int)) \xrightarrow{1/0} bool$$

It can be understood as follows. If the input list h carries 5 *potential units per element then it is sufficient to cover the cost of p_compare(h, ℓ), no potential is wasted, and* 0 *potential is left.*

Soundness That soundness theorem states that if e is well-typed in the resource type system and it evaluates to a value v then the difference between the initial and the final potential is the net resources usage. Moreover, if the potential annotations of the return value and all variables not belonging to a set $X \subseteq \text{dom}(\Gamma^r)$ are zero then e is constant-resource w.r.t X.

Theorem 1. *If* $\models E : \Gamma^r$, $E \vdash e \Downarrow v$, *and* $\Sigma^r; \Gamma^r \vdash^{q}_{q'} e : A$, *then for all* $p, r \in \mathbb{Q}^+_0$ *such that* $p = q + \Phi_E(\Gamma^r) + r$, *there exists* $p' \in \mathbb{Q}^+_0$ *satisfying* $E \vdash^{p}_{p'} e \Downarrow v$ *and* $p' = q' + \Phi(v : A) + r$.

Proof. The proof is done by induction on the length of the derivation of the evaluation judgment and the typing judgment, in which the derivation of the evaluation judgment takes priority over the typing derivation. We need to do induction on the length of both evaluation and typing derivations since on one hand, an induction of only typing derivation would fail for the case of function application, which increases the length of the typing derivation, while the length of the evaluation derivation never increases. On the other hand, if the rule C:WEAKENING is final step in the derivation, then the length of typing derivation decreases, while the length of evaluation derivation is unchanged. The additional constant r is needed to make the induction case for the let rule work. □

Theorem 2. *If* $\models E : \Gamma^r$, $E \vdash e \Downarrow v$, $\Sigma^r; \Gamma^r \vdash^{q}_{q'} e : A$, $\curlyvee(A \mid A, A)$, *and* $\forall x \in dom(\Gamma^r) \setminus X. \curlyvee(\Gamma^r(x) \mid \Gamma^r(x), \Gamma^r(x))$ *then e is constant resource w.r.t* $X \subseteq dom(\Gamma^r)$.

3.4 Type system for upper bounds

If we treat potential as an *affine* resource then we arrive that the original amortized analysis for upper bounds [51]. To this end, we allow unrestricted weakening and a relax rule in which we can waste potential.

$$\frac{\Sigma^r;\Gamma^r \vdash^{p}_{p'} e:A \qquad q \geq p \qquad q-p \geq q'-p'}{\Sigma^r;\Gamma^r \vdash^{q}_{q'} e:A}\;(\text{U:Relax}) \qquad \frac{\Sigma^r;\Gamma^r \vdash^{q}_{q'} e:B}{\Sigma^r;\Gamma^r, x:A \vdash^{q}_{q'} e:B}\;(\text{U:Weakening})$$

Additionally, we can use subtyping to waste linear potential [51]. (See the converse definition for subtyping for lower bounds below.) Similarly to Theorem 1, we can prove the following theorem.

Theorem 3. *If* $\models E:\Gamma^r$, $E \vdash e \Downarrow v$, *and* $\Sigma^r;\Gamma^r \vdash^{q}_{q'} e:A$, *then for all* $p,r \in \mathbb{Q}_0^+$ *such that* $p \geq q + \Phi_E(\Gamma^r) + r$, *there exists* $p' \in \mathbb{Q}_0^+$ *satisfying* $E \vdash^{p}_{p'} e \Downarrow v$ *and* $p' \geq q' + \Phi(v:A) + r$.

3.5 Type system for lower bounds

The type judgements for lower bounds have the same form and data types as the type judgements for constant resource usage and upper bounds. However, the intended meaning of the judgment $\Sigma^r;\Gamma^r \vdash^{q}_{q'} e:A$ is the following. Under given environment E, less than $q + \Phi_E(\Gamma)$ resource units are not sufficient to evaluate e to a value v so that more than $q' + \Phi(v:A)$ resource units are left over.

The syntax-directed typing rules are the same as the rules in constant-resource type system as given in Fig. 5. In addition, we have the structural rules in Fig. 6. The rule L:Relax is dual to U:Relax. In L:Relax, potential is treated as a *relevant* resource: We are not allowed to waste potential but we can create potential out of the blue if we ensure that we either use it or pass it to the result. The same idea is formalized for the linear potential with the subtyping rules L:Subtype and L:Supertype. The subtyping relation is defined as follows.

$$\frac{A \in \{\text{unit}, \text{bool}, \text{int}\}}{A <: A} \qquad \frac{A_1 <: A_2 \qquad p_1 \leq p_2}{L^{p_1}(A_1) <: L^{p_2}(A_2)} \qquad \frac{A_1 <: A_2 \qquad B_1 <: B_2}{A_1 * A_2 <: B_1 * B_2}$$

It holds that if $A <: B$ then $\widehat{A} = \widehat{B}$ and $\Phi(v:A) \leq \Phi(v:B)$. Suppose that it is not sufficient to evaluate e with p available resource units to get p' resource units left over. L:Subtype reflects the fact that we also cannot evaluate e with p resources get more than p' resource units after the evaluation. L:Supertype says that we also cannot evaluate e with less than p and get p' resource units afterwards.

Example. *Consider again the functions* filter_succ *and* fs_twice *given in Fig. 4 in which the resource consumption is defined using* tick *annotations. The best-case resource usage of* filter_succ(ℓ) *is* $3|\ell|+1$ *and best-case resource usage of* fs_twice(ℓ) *is* $6|\ell|+2$. *This can be reflected by the following function types for lower bounds.*

$$\begin{aligned} \text{filter\_succ}: &\quad L^3(int) \xrightarrow{1/0} L^0(int) \\ \text{fs\_twice}: &\quad L^6(int) \xrightarrow{2/0} L^0(int) \end{aligned}$$

To derive the lower bound for fs_twice, *we need the same compositional reasoning as for the derivation of the upper bound. For the inner call of* filter_succ *we use the type*

$$\text{filter\_succ}: \quad L^6(int) \xrightarrow{2/1} L^3(int).$$

It can be understood as follows. If the input list carries 6 *potential units per element then, for each element, we can either use all* 6 *(if case) or we can use* 3 *and assign* 3 *to the output (else case).*

The type system for lower bounds is a *relevant* type system [84]. That means every variable is used at least once by allowing *exchange* and *contraction properties*, but not *weakening*. However, we as in the constant-time type system we allow a restricted from of weakening if the potential annotations are zero using the rule L:Weakening. The following lemma states formally the contraction property which is derived in Fig. 7.

$$\frac{\Sigma;\Gamma \vdash^{p}_{p'} e:A \qquad q \geq p \qquad q-p \leq q'-p'}{\Sigma;\Gamma \vdash^{q}_{q'} e:A}\;(\text{L:Relax}) \qquad \frac{\Sigma;\Gamma \vdash^{q}_{q'} e:A \qquad A <: B}{\Sigma;\Gamma \vdash^{q}_{q'} e:B}\;(\text{L:Subtype})$$

$$\frac{\Sigma^r;\Gamma^r \vdash^{q}_{q'} e:B \qquad \curlyvee(A \mid A, A)}{\Sigma^r;\Gamma^r, x:A \vdash^{q}_{q'} e:B}\;(\text{L:Weakening}) \qquad \frac{\Sigma;\Gamma, x:B \vdash^{q}_{q'} e:C \qquad A <: B}{\Sigma;\Gamma, x:A \vdash^{q}_{q'} e:C}\;(\text{L:Supertype})$$

Figure 6. Structural rules for lower bounds.

$$\dfrac{\dfrac{\dfrac{\Sigma;\Gamma, x_1:A, x_2:A \vdash^{q}_{q'} e:B \qquad A_2 <: A}{\Sigma;\Gamma, x_1:A, x_2:A_2 \vdash^{q}_{q'} e:B \qquad A_1 <: A}}{\Sigma;\Gamma, x_1:A_1, x_2:A_2 \vdash^{q}_{q'} e:B} \qquad \curlyvee(A \mid A_1, A_2)}{\Sigma;\Gamma, x:A \vdash^{q}_{q'} \text{share}(x, (x_1, x_2).e):B}\;(\text{L:Contraction})$$

Figure 7. Derivation of the contraction rule for lower-bounds.

Lemma 3. *If* $\Sigma^r;\Gamma^r, x_1:A, x_2:A \vdash^{q}_{q'} e:B$ *then* $\Sigma^r;\Gamma^r, x:A \vdash^{q}_{q'}$ *share*$(x,(x_1,x_2).e):B$

The following theorems establish the soundness of the analysis. The proofs can be found in the TR [10]. Theorem 5 is proved by induction and Theorem 4 follows by contradiction.

Theorem 4. *Let* $\models E:\Gamma^r$, $E \vdash e \Downarrow v$, *and* $\Sigma^r;\Gamma^r \vdash^{q}_{q'} e:A$. *Then for all* $p,r \in \mathbb{Q}_0^+$ *such that* $p < q + \Phi_E(\Gamma^r) + r$, *there exists no* $p' \in \mathbb{Q}_0^+$ *satisfying* $E \vdash^{p}_{p'} e \Downarrow v$ *and* $p' \geq q' + \Phi(v:A) + r$.

Theorem 5. *Let* $\models E:\Gamma^r$, $E \vdash e \Downarrow v$, *and* $\Sigma^r;\Gamma^r \vdash^{q}_{q'} e:A$. *Then for all* $p,p' \in \mathbb{Q}_0^+$ *such that* $E \vdash^{p}_{p'} e \Downarrow v$ *we have* $q + \Phi_E(\Gamma^r) - (q' + \Phi(v:A)) \leq p - p'$.

3.6 Mechanization

We mechanized the soundness proofs for both the two new type systems as well as the classic AARA type system using the proof assistant Agda. The development is roughly 4000 lines of code, which includes the rules of the three types systems, the operational cost semantics, a proof of type preservation for each type system, and the soundness theorems for each type system.

One notable difference is our implementation of the typing contexts. In Agda our contexts are implemented as lists of pairs of variables and their types. Moreover, in our typing rules whenever a variable is added to the context we require a proof that the variable is fresh with respect to the existing context. This requirement is important as it allows us to preserve the invariant that the context is well formed with respect to the environment as we induct over typing and evaluation judgements in our soundness proofs. Furthermore, as our typing contexts are ordered lists we added an *exchange* rule to our typing rules.

Another important detail is in the implementation of potential. Potential $\Phi(v:A)$ for a value only is defined for well formed inputs. Inputs such as $\Phi(\texttt{nil}:\text{bool})$ are not defined. Agda is total language and as such prohibits users from implementing partial functions. Thus we require in our Agda implementation that when calculating the potential of a value of a given type the user provide a derivation that the value is well formed with respect to that type. Similarly when calculating the potential of a context, $\Phi_E(\Gamma^r)$, with respect to an environment we require that the user provide a derivation that the context is well formed with respect to that environment.

Lastly, whereas the type systems and proofs presented here used positive rational numbers, in the Agda implementation we use natural numbers. This deviation was simply due to the lacking support for rationals in the Agda standard library. By replacing a number of trivial lemmas, mostly related to associativity and commutativity, the proofs and embeddings could be transformed to use rational numbers instead.

4. A resource-aware security type system

In this section we introduce a new type system that enforces *resource-aware noninterference* to prevent the leakage of information in *high-security* variables through *low-security* channels. In addition to preventing leakage over the usual input/output information flow channels, our system incorporates the constant-resource type system discussed in Section 3 to ensure that leakage does not occur over resource side channels.

The notion of security addressed by our type system considers an attacker who wishes to learn information about certain inputs containing sensitive data by making observations of the program's public outputs and resource usage. We assume an attacker who is able to control the value of any variable she is capable of observing, and thus influence the program's behavior and resource consumption. However, in our model the attacker can only observe the program's total resource usage upon termination, and so cannot distinguish between intermediate states or between terminating and non-terminating executions.

4.1 Security types

To distinguish parts of the program under the attacker's control from those that remain secret, we annotate types with labels ranging over a lattice $(\mathscr{L}, \sqsubseteq, \sqcup, \bot)$. The elements of $\mathscr{L}$ correspond to security levels partially-ordered by $\sqsubseteq$ with a unique bottom element $\bot$. The corresponding basic security types take the form:

$$k \in \mathscr{L} \qquad S ::= (\mathrm{unit}, k) \mid (\mathrm{bool}, k) \mid (\mathrm{int}, k) \mid (L(S), k) \mid S * S$$

A security context Γ^S is a partial mapping from variable identifiers and the program counter pc to security types. The context assigns a type (unit, k) to pc to track information that may propagate through control flow as a result of branching statements. The security type for lists contains a label $L(S)$ for the elements, as well as a label k for the list's length.

As in other information flow type systems, the partial order $k \sqsubseteq k'$ indicates that the class k' is at least as restrictive as k, i.e., k is allowed to flow to k'. We assume a non-trivial security lattice that contains at least two labels: ℓ (low security) and h (high security), with $\ell \sqsubseteq h$. Following the convention defined in FlowCaml [76], we also make use of a *guard relation* $k \triangleleft S$ which denotes that all of the labels appearing in S are at least as restrictive as k. This is given in Figure 8 along with its dual notion $S \blacktriangleleft k$, called the *collecting relation*, and the standard subtyping relation $S_1 \le S_2$.

To refer to sets of variables by security class, we write $[\Gamma^S]_{\blacktriangleleft k}$ to denote the set of variable identifiers x in the domain of Γ^S such that $\Gamma^S(x) \blacktriangleleft k$, and define ${}_{k\triangleleft}[\Gamma^S]$ similarly. These gives us the set of variables upper- and lower-bounded by k, respectively. Conversely, we define $[\Gamma^S]_{\not\blacktriangleleft k} = \{x \in \mathrm{dom}(\Gamma^S) : \Gamma^S(x) \not\blacktriangleleft k\}$, the set of variables more restrictive than k. To refer to the set of variables strictly bounded below by k_1 and above by k_2, we write ${}_{k_1\triangleleft}[\Gamma^S]_{\blacktriangleleft k_2}$. Given two well-formed environments E_1 and E_2, we say that they are *k-equivalent* with respect to Γ^S, written $E_1 \equiv_k E_2$, if they agree on all variables with label at most k:

$$E_1 \equiv_k E_2 \Leftrightarrow \forall x \in [\Gamma^S]_{\blacktriangleleft k}. E_1(x) = E_2(x)$$

This relation captures the attacker's *observational equivalence* between the two environments.

$$\frac{k \sqsubseteq k' \quad T \in \mathrm{Atoms}}{k \triangleleft (T, k')} \qquad \frac{k \sqsubseteq k' \quad k \triangleleft S}{k \triangleleft (L(S), k')} \qquad \frac{k \triangleleft S_1 \quad k \triangleleft S_2}{k \triangleleft S_1 * S_2}$$

$$\frac{k' \sqsubseteq k \quad T \in \mathrm{Atoms}}{(T, k') \blacktriangleleft k} \qquad \frac{k' \sqsubseteq k \quad S \blacktriangleleft k}{(L(S), k') \blacktriangleleft k} \qquad \frac{S_1 \blacktriangleleft k \quad S_2 \blacktriangleleft k}{S_1 * S_2 \blacktriangleleft k}$$

$$\frac{k \sqsubseteq k' \quad T \in \mathrm{Atoms}}{(\mathrm{T}, k) \le (\mathrm{T}, k')} \qquad \frac{k \sqsubseteq k' \quad S \le S'}{(L(S), k) \le (L(S'), k')} \qquad \frac{S_1 \le S_1' \quad S_2 \le S_2'}{S_1 * S_2 \le S_1' * S_2'}$$

Figure 8. Guards, collecting security labels, and subtyping (Atoms = {unit, int, bool})

The first-order security types take the form:

$$\mathrm{pc} \in \mathscr{L} \qquad F^S ::= S_1 \xrightarrow{\mathrm{pc/const}} S_2 \mid S_1 \xrightarrow{\mathrm{pc}} S_2$$

The annotation pc indicates the security level of the program counter, i.e., a lower-bound on the label of any observer who is allowed to learn that a given function has been invoked. The const annotation denotes that the function body respects resource-aware noninterference. A *security signature* $\Sigma^S : \mathrm{FID} \to \wp(\mathscr{F}^S) \setminus \{\emptyset\}$ is a finite partial mapping from a set of function identifiers to a *non-empty sets* of first-order security types.

4.2 Resource-aware noninterference

We consider an adversary associated with label $k_1 \in \mathscr{L}$, who can observe and control variables in $[\Gamma^S]_{\blacktriangleleft k_1}$. Intuitively, we say that a program P satisfies resource-aware noninterference at level (k_1, k_2) with respect to Γ^S, where $k_1 \sqsubseteq k_2$, if *1)* the behavior of P does not leak any information about the contents of variables more sensitive than k_1, and *2)* does not leak any information about the contents *or sizes* of variables more sensitive than k_2. The definition follows.

Definition 2. *Let E_1 and E_2 be two well-formed environments and Γ^S be a security context sharing their domain. An expression e satisfies resource-aware noninterference at level (k_1, k_2) for $k_1 \sqsubseteq k_2$, if whenever E_1 and E_2 are:*

1. *observationally-equivalent at k_1: $E_1 \equiv_{k_1} E_2$,*
2. *size-equivalent with respect to ${}_{k_1\triangleleft}[\Gamma^S]_{\blacktriangleleft k_2}$: $E_1 \approx_{{}_{k_1\triangleleft}[\Gamma^S]_{\blacktriangleleft k_2}} E_2$*

then the following holds:

$$E_1 \left|\frac{p_1}{p_1'}\right. e \Downarrow v_1, E_2 \left|\frac{p_2}{p_2'}\right. e \Downarrow v_2 \implies v_1 = v_2 \land p_1 - p_1' = p_2 - p_2'$$

The final condition in Defintion 2 ensures two properties. First, requiring that $v_1 = v_2$ provides noninterference [38], given that E_1 and E_2 are observationally-equivalent. Second, the requirement $p_1 - p_1' = p_2 - p_2'$ ensures that the program's resource consumption will remain constant with respect to changes in variables from the set $[\Gamma^S]_{\not\blacktriangleleft k_1}$. This establishes noninterference with respect to the program's final resource consumption, and thus prevents the leakage of secret information through resource side-channels.

Before moving on, we point out an important subtlety in this definition. We require that all variables in ${}_{k_1\triangleleft}[\Gamma^S]_{\blacktriangleleft k_2}$ begin with equivalent sizes in E_1 and E_2, but not those in ${}_{k_2\triangleleft}[\Gamma^S]$. By fixing this quantity in the initial environments, we assume that an attacker is able to control and observe it, so it is not protected by the definition. This effectively establishes three classes of variables, i.e., those whose size and content are observable to the k_1-adversary, those whose size (but *not* content) is observable, and those whose size and content remain secret. In the remainder of the text, we will simplify the technical development by assuming that the third and most-restrictive class is empty, and that all of the secret variables reside in ${}_{k_1\triangleleft}[\Gamma^S]_{\blacktriangleleft k_2}$.

Assumptions and limitations. The definition of resource-aware noninterference given in Definition 2 assumes an adversary whose observations of resource consumption match the cost semantics given in Section 3. Depending on how the costs are parameterized, this may not match the reality of actual resource use in a physical environment on modern hardware. For example, if the processor's instruction cache is not accounted for then this may introduce an exploitable discrepancy between the guarantees provided by the type system and the real-world attacker's observations [22, 40, 70]. In this work, we use a cost semantics that is conceptually straightforward, and leave as future work the development of more precise models (such as the one described in by Zhang et al. [88]) that are faithful to the subtleties of hardware platforms.

4.3 Proving resource-aware noninterference

There are two extreme ways of proving resource-aware noninterference. Assume we already have established classic noninterference by using an information-flow type system. The first way is to additionally prove constant resource usage *globally* by forgetting the security labels and showing that the program has constant resource usage. This is a sound approach but it requires us to reason about parts of the programs that are not affected by secret data. It would therefore result the rejection of programs that have the resource-aware noninterference property but are not constant resource. The second way is to prove constant resource usage *locally* by ensuring that every conditional that branches on secret values is constant time. However, this local approach is problematic because it is not compositional. Consider the following examples in which *rev* is the standard reverse function.

```
let f1(b,x) =
  let z = if b then x else [] in rev z

let f2(b,x,y) =
  let z = if b then let _ = rev y in x
          else let _ = rev x in y
  in rev z
```

If we assume a cost model in which we count the number of function calls then the cost of *rev(x)* is $|x|$. So *rev* is constant resource w.r.t its argument. Moreover, the expression *if b then x else []* is constant resource. However, *f1* is not constant resource. In contrast, the conditional in the function *f2* is not constant resource. However, *f2* is a constant resource function. The function *f2* can be automatically analyzed with the constant-resource type system from Section 3 while *f1* is correctly rejected.

The idea of our type system for resource-aware noninterference is to allow both global and local reasoning about resource consumption as well as arbitrary intermediate levels. *We ensure that every expression that is typed in a high security context is part of a constant resource expression.* In this way, we get the benefits of local reasoning without loosing compositionality.

4.4 Typing rules and soundness

We combine our type system for constant resource usage with a standard information flow type system which based on FlowCaml [72]. The interface between the two type systems is relatively light and the idea is applicable to other methods for proving constant resource use as well as other security type systems.

In the type judgement, an expression is typed under a type context Γ^s and a label pc. The pc label can be considered an upper bound on the security labels of all values that affect the control flow of the expression and a lower bound on the labels of the function's effects [72]. As mentioned earlier, we will simplify the technical development by assuming that the third and most-restrictive class is empty, and that all of the secret variables reside in ${}_{k_1\triangleleft}[\Gamma^s]_{\blacktriangleleft k_2}$, that is, the typing rules here guarantee that well-typed expressions provably satisfy the resource-aware noninterference property w.r.t. changes in variables from the set $[\Gamma^s]_{\not\blacktriangleleft k_1}$. We define two type judgements of the following form, in which we write const(e) if there exists Γ^r and Σ^r such that $\Sigma^r;\Gamma^r \,\Big|\frac{q}{q'}\; e : A$, $\curlyvee(A \mid A, A)$, and $\forall x \in [\Gamma^s]_{\not\blacktriangleleft k_1}.\ \curlyvee(\Gamma^r(x) \mid \Gamma^r(x), \Gamma^r(x))$.

$$\mathrm{pc}; \Sigma^s; \Gamma^s \vdash^{\mathrm{const}} e : S \quad \text{and} \quad \mathrm{pc}; \Sigma^s; \Gamma^s \vdash e : S\,.$$

The judgement with the const annotation states that under a security configuration given by Γ^s and the label pc, e has type S and it satisfies resource-ware noninterference w.r.t. changes in variables from the set $[\Gamma^s]_{\not\blacktriangleleft k_1}$. The second judgement indicates that e satisfies the noninterference property but does not make any guarantees about resource-based side channels. Selected typing rules are given in Fig. 9. We implicitly assume that the security types and the resource-annotated counterparts have the same base types.

Note that the standard information flow typing rules [45, 72] can be obtained by removing the const annotation from all judgements. Consider for instance the rule SR:IF for conditional expressions. By executing the true or false branches, an adversary could gain information about the conditional value whose security label is k_x. Therefore the conditional expression must be type-checked under a security assumption at least as restrictive as pc and k_x. This is a standard requirement in any information flow type system. In the following we will focus on explaining how the rules restrict the observable resource usage instead of these classic noninterference aspects.

The most interesting rules are SR:C-GEN and the rules for and let expressions and conditionals, which block leakage over resource usage when branching on high security data. SR:C-GEN allows us to *globally* reason about constant resource usage for an arbitrary subexpression that has the noninterference property. For example, we can apply SR:IF, the standard rule for conditionals, first and then SR:C-GEN to prove that the expression is constant resource. Alternatively, we can use rules such as SR:L-IF and SR:L-LET to *locally* reason about resource use.

The rule SR:L-LET reflects the fact that if both e_1 and e_2 have the resource-aware noninterference property and the size of x only depends on low security data then let$(x, e_1, x.e_2)$ has the resource-aware noninterference property. The reasoning is similar for rule SR:L-IF where we require that the variable x does not depend on high security data.

Leaf expressions such as $\mathrm{op}_\diamond(x_1, x_2)$ and $\mathrm{cons}(x_h, x_t)$ have constant resource usage. Thus their judgments are always associated with the qualifier const as shown in the rule SR:B-OP. The rule SR:C-FUN states that if a function's body has the resource-aware noninterference property then the function application has the resource-aware noninterference property too. If the argument's label is low security data, bounded below by k_1, then the function application has the resource-aware noninterference property since the value of the argument is always the same under any k-equivalent environments. It is reflected by rule SR:L-ARG.

Example. *Recall functions compare and p_compare in Fig. 3. Suppose the content of the first list is secret and the length is public. Thus it has type $(L(int, h), \ell)$. While the second list controlled by adversaries is public, hence it has type $(L(int, \ell), \ell)$. Assume that the pc label is ℓ and $[\Gamma^s]_{\not\blacktriangleleft k_1} = [\Gamma^s]_{\not\blacktriangleleft \ell}$. The return value's label depends on the content of the first list elements whose label is h. Thus it must be assigned the label h to make the functions well-typed.*

$$\begin{aligned} \mathsf{compare} : &\quad ((L(int,h),\ell),(L(int,\ell),\ell)) \xrightarrow{\ell} (bool,h) \\ \mathsf{p\_compare} : &\quad ((L(int,h),\ell),(L(int,\ell),\ell)) \xrightarrow{\ell/const} (bool,h) \end{aligned}$$

Here, both functions satisfy the noninterference property at security label ℓ. However, only p_compare is resource-aware noninterference function w.r.t $[\Gamma^s]_{\not\blacktriangleleft \ell}$, or the secret list.

$$\text{(SR:B-OP)}\quad \frac{x_1 : (\text{bool}, k_{x_1}) \in \Gamma^s \qquad x_2 : (\text{bool}, k_{x_2}) \in \Gamma^s \qquad \text{pc} \sqsubseteq k_{x_1} \sqcup k_{x_2} \qquad \diamond \in \{\mathsf{and}, \mathsf{or}\}}{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} \text{op}_\diamond(x_1, x_2) : (\text{bool}, k_{x_1} \sqcup k_{x_2})}$$

$$\text{(SR:GEN)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} e : S}{\text{pc}; \Sigma^s; \Gamma^s \vdash e : S}$$

$$\text{(SR:C-GEN)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \vdash e : S \qquad \text{const}(e)}{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} e : S}$$

$$\text{(SR:L-ARG)}\quad \frac{x : S_1 \in \Gamma^s \qquad \Sigma^s(f) = S_1 \xrightarrow{\text{pc}'} S_2 \qquad \text{pc} \sqsubseteq \text{pc}' \qquad S_1 \blacktriangleleft k_1}{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} \text{app}(f, x) : S_2}$$

$$\text{(SR:C-FUN)}\quad \frac{\Sigma^s(f) = S_1 \xrightarrow{\text{pc}'/\text{const}} S_2 \qquad x : S_1 \in \Gamma^s \qquad \text{pc} \sqsubseteq \text{pc}'}{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} \text{app}(f, x) : S_2}$$

$$\text{(SR:FUN)}\quad \frac{\Sigma^s(f) = S_1 \xrightarrow{\text{pc}'} S_2 \qquad x : S_1 \in \Gamma^s \qquad \text{pc} \sqsubseteq \text{pc}'}{\text{pc}; \Sigma^s; \Gamma^s \vdash \text{app}(f, x) : S_2}$$

$$\text{(SR:C-SUBTYPING)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} e : S \qquad S \le S'}{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} e : S'}$$

$$\text{(SR:IF)}\quad \frac{x : (\text{bool}, k_x) \in \Gamma^s \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash e_t : S \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash e_f : S \qquad \text{pc} \sqcup k_x \lhd S}{\text{pc}; \Sigma^s; \Gamma^s \vdash \text{if}(x, e_t, e_f) : S}$$

$$\text{(SR:L-IF)}\quad \frac{\text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash^{\text{const}} e_t : S \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash^{\text{const}} e_f : S \qquad x : (\text{bool}, k_x) \in \Gamma^s \qquad \text{pc} \sqcup k_x \lhd S \qquad k_x \sqsubseteq k_1}{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} \text{if}(x, e_t, e_f) : S}$$

$$\text{(SR:SUBTYPING)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \vdash e : S \qquad S \le S'}{\text{pc}; \Sigma^s; \Gamma^s \vdash e : S'}$$

$$\text{(SR:L-LET)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} e_1 : S_1 \qquad \text{pc}; \Sigma^s; \Gamma^s, x : S_1 \vdash^{\text{const}} e_2 : S_2 \qquad S_1 \blacktriangleleft k_1}{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} \text{let}(x, e_1, x.e_2) : S_2}$$

$$\text{(SR:LET)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \vdash e_1 : S_1 \qquad \text{pc}; \Sigma^s; \Gamma^s, x : S_1 \vdash e_2 : S_2}{\text{pc}; \Sigma^s; \Gamma^s \vdash \text{let}(x, e_1, x.e_2) : S_2}$$

$$\text{(SR:MATCH-L)}\quad \frac{x : (L(S), k_x) \in \Gamma^s \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash e_1 : S_1 \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s, x_h : S, x_t : (L(S), k_x) \vdash e_2 : S_1 \qquad \text{pc} \sqcup k_x \lhd S_1}{\text{pc}; \Sigma^s; \Gamma^s \vdash \text{match}(x, e_1, (x_h, x_t).e_2) : S_1}$$

$$\text{(SR:C-MATCH-L)}\quad \frac{x : (L(S), k_x) \in \Gamma^s \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash^{\text{const}} e_1 : S_1 \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s, x_h : S, x_t : (L(S), k_x) \vdash^{\text{const}} e_2 : S_1 \qquad \text{pc} \sqcup k_x \lhd S_1}{\text{pc}; \Sigma^s; \Gamma^s \vdash^{\text{const}} \text{match}(x, e_1, (x_h, x_t).e_2) : S_1}$$

Figure 9. Selected security typing rules

Example. *Consider the following function cond_rev in which rev is the standard reverse function.*

```
let cond_rev(l1,l2,b1,b2) =
  if b1 then let r =
    if b2 then rev l1; l2 else rev l2; l1
    in rev r; ()
  else ()
```

Assume that l_1, l_2, b_1 and b_2 have types $(L(int, h), \ell)$, $(L(int, h), \ell)$, $(bool, \ell)$, and $(bool, h)$, respectively. Given the rev function is constant w.r.t the argument, the inner if is not resource-aware noninterference. However, the let expression is resource-aware noninterference w.r.t $[\Gamma^s]_{\blacktriangleleft \ell} = \{l_1, l_2, b_2\}$ by applying the rule SR:C-GEN. *Finally, the outer if branching on low security data and its branches of are resource-aware noninterference, has resource-aware noninterference property w.r.t $\{l_1, l_2, b_2\}$ at level ℓ by the rule* SR:L-IF. *We obtain the following inferred type.*

$$\text{cond\_rev} : \quad ((L(int, h), \ell), (L(int, h), \ell), (bool, \ell), (bool, h)) \xrightarrow{\ell / const} (unit, \ell)$$

We now prove the soundness of the type system w.r.t the definition of resource-aware noninterference. The soundness theorem states that if e is well-typed expression with the const annotation then it is resource-aware noninterference expression at level k_1.

The following two lemmas are needed in the soundness proof. The first lemma states that the type system satisfies the standard *simple security* property [83] and the second shows that the type system prove classic noninterference.

Lemma 4. *Let $pc; \Sigma^s; \Gamma^s \vdash e : S$ or $pc; \Sigma^s; \Gamma^s \vdash^{const} e : S$. For all variables x in e, if $S \blacktriangleleft k_1$ then $\Gamma^s(x) \blacktriangleleft k_1$.*

Lemma 5. *Let $pc; \Sigma^s; \Gamma^s \vdash e : S$ or $pc; \Sigma^s; \Gamma^s \vdash^{const} e : S$, $E_1 \vdash e \Downarrow v_1$, $E_2 \vdash e \Downarrow v_2$, and $E_1 \equiv_{k_1} E_2$. Then $v_1 = v_2$ if $S \blacktriangleleft k_1$.*

Theorem 6. *If $\models E : \Gamma^s$, $E \vdash e \Downarrow v$, and $pc; \Sigma^s; \Gamma^s \vdash^{const} e : S$ then e is resource-aware noninterference expression at level k_1.*

Proof. The proof is done by induction on the structure of the typing derivation and the evaluation derivation. Let X be the set of variables $[\Gamma^s]_{\blacktriangleleft k_1}$. For all environments E_1, E_2 such that $E_1 \approx_X E_2$ and $E_1 \equiv_{k_1} E_2$, if $E_1 \vdash^{p_1}_{p_1'} e \Downarrow v_1$ and $E_2 \vdash^{p_2}_{p_2'} e \Downarrow v_2$. We then show that $p_1 - p_1' = p_2 - p_2'$ and $v_1 = v_2$ if $S \blacktriangleleft k_1$. We illustrate one case of the conditional expression. Suppose e is of the form $\text{if}(x, e_t, e_f)$, thus the typing derivation ends with an application of either the rule SR:L-IF or SR:C-GEN. By Lemma 5, if $S \blacktriangleleft k_1$ then $v_1 = v_2$.

- Case SR:L-IF. By the hypothesis we have $E_1(x) = E_2(x)$. Assume that $E_1(x) = E_2(x) = \mathsf{true}$, by the evaluation rule E:IF-TRUE, $E_1 \vdash^{p_1 - K^{\text{cond}}}_{p_1'} e_t \Downarrow v_1$ and $E_2 \vdash^{p_2 - K^{\text{cond}}}_{p_2'} e_t \Downarrow v_2$. By induction for e_t we have $p_1 - p_1' = p_2 - p_2'$. It is similar for $E_1(x) = E_2(x) = \mathsf{false}$.
- Case SR:C-GEN. Since $E_1 \approx_X E_2$ w.r.t Γ^s, we have $E_1 \approx_X E_2$ w.r.t Γ^r. By the hypothesis we have $\text{const}(e)$. Thus by Theorem 2, it follows $p_1 - p_1' = p_2 - p_2'$. □

5. Quantifying and transforming out leakages

We present techniques to quantify the amount of information leakage through resource usage and transform leaky programs into constant resource programs. The quantification relies on the lower and upper bounds inferred by our resource type systems. The transformation pads the programs with dummy computations so that the evaluations consume the same amount of resource usage and the outputs are identical with the original programs. In the current implementation, these dummy computations are added into programs by users and the padding parameters are automatically added by our analyzer to obtain the optimal values. It would be straightforward to make the process fully automatic but the interactive flavor of our approach helps to get a better understanding of the system.

5.1 Quantification

Recall from Section 4 that we assume an adversary at level k_1 who is always able to observe *1)* the values of variables in $[\Gamma^s]_{\blacktriangleleft k_1}$, and

2) the final resource consumption of the program. For many programs, it may be the case that changes to the secret variables $[\Gamma^s]_{\not\blacktriangleleft k_1}$ effect observable differences in the program's final resource consumption, but only allow the attacker to learn partial information about the corresponding secrets. In this section, we show that the upper and lower-bound information provided by our type systems allow us to derive bounds on the amount of partial information that is leaked.

To quantify the amount of leaked information, we measure the number of distinct environments that the attacker could deduce as having produced a given resource consumption observation. However, becuase there may be an unbounded number of such environments, we parameterize this quantity on the size of the values contained in each environment. Let $\mathbf{E}^N$ denote the space of environments with values of size characterized by N. Given an environment E and expression e, define $U(E,e) = p_\delta$ such that $E \,\Big|^{\,p}_{\,p'}\, e \Downarrow v$ and $p_\delta = p - p'$. Then for an expression e and resource observation p_δ, we define the set $R_N(e, p_\delta)$ which captures the attacker's uncertainty about the environment which produced p_δ:

$$R_N(e, p_\delta) = \{E' \in \mathbf{E}^N : U(E,e) = p_\delta\}$$

Notice that when $|R_N(e,p)| = 1$, the attacker can deduce exactly which environment was used, whereas when this quantity is large little additional information is learned from p_δ. This gives us a natural definition of leakage, which is obtained by aggregating the inverse of the cardinality of R_N over the possible initial environments of e:

$$C_N(e) = \left(\sum_{E \in \mathbf{E}^N} \frac{1}{|R_N(e, U(E,e))|} \right) - 1$$

$C_N(e)$ corresponds to our intuition about leakage. When e leaks no information through resource consumption, then each term in the summation will be $1/|\mathbf{E}^{sizes}|$ giving $C_N(e) = 0$, whereas if e leaks perfect information about its starting environment then each term will be 1, leading to $C_N(e) = |\mathbf{E}^N| - 1$.

Theorem 7. *Let P^e_N be the complete set of resource observations producible by expression e under environments of size N, i.e.,*

$$P^e_N = \{p : \exists E \in \mathbf{E}^N . U(E,e) = p\}$$

Then $|P^e_N| = C_N(e) + 1$.

Lemma 6. *Let $l_e(N)$ and $u_e(N)$ be lower and upper-bounds on the resource consumption of e for inputs of size N. If $U(E,e) \in \mathbb{Z}$ for all environments E, then $C_N(e) \le u_e(N) - l_e(N)$.*

Lemma 7. *Assume that environments are sampled uniform-randomly from $\mathbf{E}^N$. Then the Shannon entropy of P^e_N is given by $C_N(e)$: $H(P^e_N) \le \log_2(C_N(e) + 1)$*

Lemma 6 leverages Theorem 7 to derive an upper-bound on leakage from upper and lower-bounds on resource usage. This result only holds when the possible resource observations of e are integral, as this ensures that the interval $[l_e(N), u_e(N)] \supseteq P^e_N$ is finite. Lemma 7 relates $C_N(e)$ to Shannon entropy, which is commonly used to characterize information leakage [59, 60, 88].

5.2 Transformation

To transform programs into constant resource programs we extend the type system for constant resource use from Section 3. Recall that the type system treats potential in a linear fashion to ensure that potential is not wasted. We will now add *sinks* for potential which will be able to absorb excess potential. At runtime the the sinks will consume the exact amount of resources that have been statically-absorbed to ensure that potential is still treated in a linear way. The advantage of this approach is that the worst-case resource consumption is often not affected by the transformation. Additionally, we do not need to keep track of resource usage at runtime to pad the resource usage at the sinks, because the amount of resource that must be discarded is statically-determined by the type system. Finally, we automatically obtain a type derivation that serves as a proof that the transformation is constant-resource.

More precisely, the sinks are represented by the syntactic form: $\text{consume}_{(\mathrm{A},p)}(x)$. Here, A is a resource-annotated type and $p \in \mathbb{Q}_{\ge 0}$ is a non-negative rational number. The idea is that A and p define the resource consumption of the expression. In the implementation, the user only has to write $\text{consume}(x)$, and the annotations are added via automatic syntax elaboration during the resource type inference.

Let E be a well-formed environment w.r.t Γ^r. For every $x \in \text{dom}(\Gamma)$ with $\Gamma^r(x) = A$, the expression $\text{consume}_{(\mathrm{A},p)}(x)$ consumes $\Phi(E(x) : A) + p$ resource units and evaluate to $()$. The evaluation and typing rules for sinks are:

$$\frac{}{\Sigma^r; x{:}A \,\Big|^{\,p}_{\,0}\, \text{consume}_{(A,p)}(x) : \text{unit}}\ (\text{A:Consume}) \qquad \frac{q = q' + \Phi(E(x) : A) + p}{E \,\Big|^{\,q}_{\,q'}\, \text{consume}_{(\mathrm{A},p)}(x) \Downarrow ()}\ (\text{E:Consume})$$

The extension of the proof of Theorem 1 to consume expressions is straightforward.

Adding consume expressions Let e_i be a subexpression of e and let e'_i be the expression $\text{let}(z, \text{consume}(x_1, \cdots, x_n), z.e_i)$ for some variables x_i. Let e' be the expression obtained from e by replacing e_i with e'_i. We write $e \hookrightarrow e'$ for such a transformation. Note that additional *share* and *let* expressions have to be added to convert e'_i into share-let normal form.

Lemma 8. *If $\Sigma; \Gamma \vdash e : T$, $E \vdash e \Downarrow v$, and $e \hookrightarrow e'$ then $\Sigma; \Gamma \vdash e' : T$ and $E \vdash e' \Downarrow v$.*

To transform an expression e into a constant resource expressions we perform multiple transformations $e \hookrightarrow e'$ which do not affect the type and semantics of e. This can be done automatically but in our implementation it works in an interactive fashion, meaning that users are responsible for the locations where consume expressions are put. The analyzer will infer the annotations A and constants p of the given consume expressions during type inference. If the inference is successful then we have $\text{const}(e')$ for the transformed program e'.

Example. *Recall the function compare form Fig. 3. To turn compare into a constant resource function. We insert consume expressions as shown below. Users can insert many consume expressions and the analyzer will determine which consume the are actually needed.*

```
let rec c_compare (h,l) = match h with
  | [] → (match l with | [] → Raml.tick 1.0; true
    | y::ys → Raml.tick 1.0; false)
  | x::xs → match l with
    | [] → Raml.tick 1.0; Raml.consume xs; false
    | y::ys → if (x = y) then
        Raml.tick 5.0; c_compare (xs,ys)
      else Raml.tick 5.0; Raml.consume xs; false
```

We automatically obtain the following typing of the transformed function and the consume expressions:

$$\begin{array}{lll} \text{c\_compare :} & (L^5(int), L^0(int)) \xrightarrow{1/0} bool & \\ \text{consume :} & L^5(int) \xrightarrow{5/0} unit & (at\ line\ 5) \\ \text{consume :} & L^5(int) \xrightarrow{1/0} unit & (at\ line\ 8) \end{array}$$

The worst-case resource consumption of the unmodified function c_compare h l is $1 + 5|h|$. Thus the consumption of the first consume must be $5 + 5(|h| - 1 - |\ell|)$ when h is longer than l. Otherwise, the consumption is zero. The second one consumes $1 + 5(|h_1| - 1)$, where h_1 is the sub-list of h from the first node which is different from the corresponding node in l.

6. Implementation and Evaluation

Type Inference The type inference for the type systems for constant resource and lower bounds are implemented in RAML [9]. RAML is integrated in Inria's OCaml compiler and supports polynomial bounds, user-defined inductive types, higher-order functions, polymorphism, and arrays and references. All features are implemented for the new type systems and are basically orthogonal to the new ideas that we explained in the simplified setting of this article. The implementation is publically available as source code and in an easy-to-use web interface [46].

The type inference is technically similar to the inference of upper bounds [51]. We first integrate the structural rules of the respective type system in the syntax directed rules. For example, weakening and relaxation is applied at branching points such as conditional and pattern matching. We then compute a type derivation in which all resource annotations are replace by (yet unknown) variables. For each type rule we produce a set of linear constraints that specify the properties of valid annotations. These linear constraints are then solved by the LP solver CLP to obtain a type derivation in which the annotations are rational numbers.

An interesting challenge lies in finding a solution for the linear constraints that leads to the best bound for a given function. For upper bounds, we simply disregard the potential of the result type and provide an objective function that minimizes the annotations of the arguments. The same strategy works the constant-time type systems. An interesting property is that the solution to the linear program is unique if we require that the potential of the result type is zero. To obtain the optimal lower bound we want to maximize the potential of the arguments and minimize the potential of the result. We currently simply maximize the potential of the arguments while requiring the potential of the result to be zero. Another approach would be to first minimize the output potential and then maximize the input potential.

Resource-aware noninterference We are currently integrating our constant-time type system with FlowCaml [76]. The combined inference is based on the typing rules in Fig. 9. It is possible to derive a set of type inference rules in the same way as for FlowCaml [72, 78]. One of the challenges in the integration is interfacing FlowCaml's type inference with our constant-time type system in rule SR:C-GEN. In the implementation, we intend for each application of SR:C-GEN to generate an intermediate representation of the expression in RAML for the expression under consideration, in which all types are annotated with fresh resource annotations along with the set of variables X. The expression is marked with the qualifier const if a RAML can prove that it is constant time. The type inference algorithm always tries to apply the syntax-directed rules first before using SR:C-GEN.

Evaluation Table 1 shows the verification and computation of constant resource usage, lower and upper bounds for number of functions with different size in terms of number of line of code (LOC). The cost models are specified by several different cost metric, i.e., number of evaluation steps, number of multiplication operations. Note that the computed upper bounds are also the resource usages of functions which are padded using *consume* expressions. The experiments were run on machine with Intel Core i5 2.4 GHz processor and 8GB RAM under the OS X 10.11.5. The run-time of the analysis of varies from 0.03 to 14.34 seconds depending on the function code's complexity. The example programs that we analyzed consist of commonly-used primitives (`cond_rev`, `trunc_rev`, `compare`, `find`, `filter`), functions related to cryptography (`tea_enc`, `tea_dec`, `rsa`), and examples taken from Haeberlan et al. [42] (`ipquery`, `kmeans`). The full source code of the examples can be found in the technical report [10].

The encryption functions `tea_enc` and `tea_dec` correspond to the encryption and decryption routines of the Corrected Block Tiny Encryption Algorithm [86], a block cipher presented by Needham and Wheeler in an unpublished technical report in 1998. Our implementation correctly identifies these operations as constant-time in the number of primitive operations performed. We applied this cost model for these examples due to the presence of bitwise operations in the original algorithm, which are not currently supported in RAML. In order to derive a more meaningful bound, we implemented bitwise operations in the example source and counted them as single operations.

The two examples taken from Haeberlen et al. [42] were originally created in a study of timing attacks in differentially-private data processing systems. `ipquery` applies pattern matching to a database derived from Apache server logs, counting the number of matches and non-matches. `kmeans` implements the k-means clustering algorithm [64], which partitions a set of geometric points into k clusters that minimize the total inter-cluster distance between points. Haeberlen et al. demonstrated that when a query applied to a dataset introduces attacker-observable timing variations, then the privacy guarantees provided by differential privacy are negated. To address this, they proposed a mitigation approach that enforces constant-time behavior by aborting or padding the query's runtime. Our implementation is able to determine that these queries were constant-time to begin with, and thus did not need black-box mitigation.

7. Related work

Resource bounds Our work builds on past research on automatic amortized resource analysis (AARA). AARA has been introduced by Hofmann and Jost for a strict first-order functional language with built-in data types to derive linear heap-memory bounds [51]. It has then been extended to polynomial bounds [47, 48, 50] for strict and higher-order [9, 57] functions. AARA has also been used to derive linear bounds for lazy functional programs [75, 82] and object-oriented programs [52, 55]. In another line of work, the technique has been integrated into separation logic [12] to derive bounds that depend on mutable data-structures, and into Hoare logic to derive linear bounds that depend on integers [25, 26]. The potential method of amortized analysis has also been used to manually verify the complexity of algorithms and data-structures using proof assistants [28, 67].

As discussed in the introduction, AARA has been successfully extended to other resources and language features [12, 23, 52, 55, 57, 75, 82] and to polynomial bounds [47, 49, 50, 53, 54]. Amortized analysis has also been used to verify bounds on algorithms and data structures with proof assistants [28, 67]. In contrast to our work, these techniques can only derive upper bounds and prove constant resource consumption. This focus on upper bounds is shared with automatic resource analysis techniques that based on sized types [80, 81], linear dependent types [62, 63], and other type systems [29, 31, 32]. Similarly, semiautomatic analyses [14, 17, 33, 39] focus on upper bounds too.

Automatic resource bound analysis is also actively studied for imperative languages using recurrence relations [4, 7, 36] and abstract interpretation [18, 27, 41, 77, 90]. While these techniques focus on worst-case bounds, it is possible to use similar techniques for deriving lower bounds [3]. The advantage of our method is that it is compositional, deal well with amortization effects, and works for language features such as user-defined data types and higher-order functions. Another approach to (worst-case) bound analysis is based on techniques from term rewriting [13, 20, 68], which mainly focus on upper bounds. One line of work [37] derives lower bounds on the *worst-case* behavior of programs which is different from our lower bounds on the best-case behavior.

Side channels Analyzing and mitigating potential sources of side channel leakage is an increasingly well-studied area. Sev-

Constant Function	LOC	Metric	Resource Usage	Time
cond_rev : $(L(\text{int}), L(\text{int}), \text{bool}) \to \text{unit}$	20	steps	$13n{+}13x{+}35$	0.03s
trunc_rev : $(L(\text{int}), \text{int}) \to L(\text{int})$	28	function calls	$1n$	0.06s
ipquery : $L(\text{logline}) \to (L(\text{int}), L(\text{int}))$	86	steps	$86n{+}99$	0.86s
kmeans : $L(\text{float}, \text{float}) \to L(\text{float}, \text{float})$	170	steps	$1246n{+}3784$	8.18s
tea_enc : $(L(\text{int}), L(\text{int}), \text{nat}) \to L(\text{int})$	306	ticks	$128n^2z{+}32nxz{+}1184nz{+}96n{+}128z{+}96$	13.73s
tea_dec : $(L(\text{int}), L(\text{int}), \text{nat}) \to L(\text{int})$	306	ticks	$128n^2z{+}32nxz{+}1184nz{+}96n{+}96z{+}96$	14.34s

Function	LOC	Metric	Lower Bound	Time	Upper Bound	Time
compare : $(L(\text{int}), L(\text{int})) \to \text{bool}$	60	steps	7	0.05s	$16n{+}7$	0.09s
find : $(L(\text{int}), \text{int}) \to \text{bool}$	40	steps	5	0.04s	$14n{+}5$	0.02s
rsa : $(L(\text{bool}), \text{int}, \text{int}) \to \text{int}$	42	multiplications	$1n$	0.07s	$2n$	0.05s
filter : $L(\text{int}) \to L(\text{int})$	30	steps	$13n{+}5$	0.05s	$20n{+}5$	0.04s
isortlist : $L(L(\text{int})) \to L(L(\text{int}))$	60	steps	$21n{+}5$	0.13s	$12n^2{+}9n{+}10n^2m{-}10nm{+}5$	0.43s
bfs_tree : $(\text{btree}, \text{int}) \to \text{btree option}$	116	steps	15	0.30s	$92n{+}24$	0.32s

Table 1. The computed resource usage in case of constant function, the computed lower and upper bounds, and the run-time of the analysis in seconds. Note that constant resource usage, lower and upper bounds are the same when a function is constant. In the computed resource usage n is the size of the first argument, $m = \max_{1 \le i \le n} m_i$ where m_i are the sizes the first argument's elements, x is the size of the second argument, $y = \max_{1 \le i \le n} y_i$ where y_i are the sizes the second argument's elements, and z is the value of the third argument.

eral groups have proposed using type systems or other program analyses to transform programs into constant-time versions by padding out branches and loops with "dummy" commands [1, 15, 30, 44, 66, 88]. Because these systems do not account for timing explicitly, as is the case for our work, this approach will in nearly all cases introduce an unnecessary performance penalty. The most recent of these systems by Zhang et al. [88] describes an approach for mitigating side channels using a combination of security types, hardware assistance, and predictive mitigation [87]. Unlike the type system given in Section 4, theirs does not guarantee that information is not leaked through timing. Rather, they show that the amount of this leakage is bounded by the variation of the mitigation commands.

Köpf and Basin [59] presented an information-theoretic model for adaptive side channel attacks that occur over multiple runs of a program, as well as an automated analysis for measuring the corresponding leakage. Because their analysis is doubly-exponential in the number of steps taken by the attacker, they describe an approximate version based on a greedy heuristic. Mardziel et al. later generalized this model to probabilistic systems [65], secrets that change over time, and wait-adaptive adversaries. Pasareanu et al. [71] proposed a symbolic approach for the multi-run setting based on MaxSAT and model counting. Doychev et al. [34] and Köpf et al. [60] consider cache side channels, and present analyses that over-approximate leakage using model-counting techniques. While these analyses are sometimes able to derive useful bounds on the leakage produced by binaries on real hardware, they do not incorporate security labels to distinguish between different sources, and were not applied to verifying constant-time behavior.

FlowTracker [73] and ct-verif [6] are both constant-time analyses built on top of LLVM which reason about timing and other side-channel behavior indirectly through control and address-dependence on secret inputs. VirtualCert [16] instruments CompCert with a constant-time analysis based on similar reasoning about control and address-dependence. These approaches are intended for code that has been written in "constant-time style", and thus impose effective restrictions on the expressiveness of the programs that they will work on. Because our approach reasons about resources explicitly, it imposes no a priori restrictions on program expressiveness.

Information flow A long line of prior work looks at preventing information flows using type systems. Sabelfeld and Myers [74] present an excellent overview of much of the early work in this area. The work most closely related to our security type system is FlowCaml [72], which provides a type system that enforces noninterference for a core of ML with references, exceptions, and let-polymorphism. The portion of our type system that applies to traditional noninterference coincides with the rules used in FlowCaml. However, the rules in our type system are not only designed to track flows of information, but they are also used to incorporate the information flow and resource usage behavior such as the rules SR:L-If and SR:L-Let. Moreover, our type system constructs a flexible interface between FlowCaml and the constant resource type system for reasoning about resource consumption, meaning that the rules can be easily adapted to integrate into any information flow type system.

The primary difference between our work and the prior work on information flow type systems is best summarized in terms of our attacker model. Whereas prior work assumes an attacker that can manipulate low-security inputs and observe low-security outputs, our type system enhances this attacker by granting the ability to observe the program's final resource consumption. This broadens the relevant class of attacks to include resource side channels, which we prevent by extending a traditional information flow type system with explicit reasoning about the resource behavior of the program using AARA.

8. Conclusion

We have introduced new substructural type systems for automatically deriving lower bounds and proving constant resource usage. The evaluation with the implementation in RAML shows that the technique extends beyond the core language that we study in this paper and works for realistic example programs. We have shown how the new type systems can interact with information-flow type systems to prove resource-aware noninterference. Moreover, the type system for constant resource can be used to automatically remove side-channel vulnerabilities from programs.

There are many interesting connection between security and (automatic) quantitative resource analysis that we plan to study in the future. Two concrete projects that we already started are the integration of the type systems for upper and lower bounds with information flow type systems to precisely quantify the resource-based information leakage at certain security levels. Another direction is to more precisely characterize the amount of information that can be obtained about secretes by making one particular resource-usage observation.

References

[1] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, 2000.

[2] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román, and D. Zanardini. Termination and Cost Analysis with COSTA and its User Interfaces. *Electr. Notes Theor. Comput. Sci.*, 258(1):109–121, 2009.

[3] E. Albert, S. Genaim, and A. N. Masud. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Transactions on Computational Logic*, 14(3), 2013.

[4] E. Albert, J. C. Fernández, and G. Román-Díez. Non-cumulative Resource Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15)*, 2015.

[5] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[6] J. C. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (to appear)*, Austin, TX, Aug. 2016.

[7] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *19th Int. Static Analysis Symp. (SAS'12)*, 2012.

[8] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, 2015.

[9] Anonymous. Towards Automatic Resource Bound Analysis for OCaml. Technical Report TR-??-???, Removed for double blind reviewing, 2015.

[10] Anonymous. Quantifying and Preventing Side Channels with Substructural Type Systems. Technical Report TR-??-???, Removed for double blind reviewing. Submitted as anonymous auxiliary material, 2016.

[11] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[12] R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, 2010.

[13] M. Avanzini and G. Moser. A Combination Framework for Complexity. In *24th International Conference on Rewriting Techniques and Applications (RTA'13)*, 2013.

[14] M. Avanzini, U. D. Lago, and G. Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

[15] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153(2), May 2006.

[16] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, 2014.

[17] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.

[18] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*, 2010.

[19] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th International Conference on World Wide Web*, 2007.

[20] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *20th Int. Conf. on Tools and Alg. for the Constr. and Anal. of Systems (TACAS'14)*, 2014.

[21] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Annual USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[22] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Computer Networks*, 2005.

[23] B. Campbell. Amortised Memory Analysis using the Depth of Data Structures. In *18th Euro. Symp. on Prog. (ESOP'09)*, 2009.

[24] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux. *Password Interception in a SSL/TLS Channel*. Berlin, Heidelberg, 2003.

[25] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-End Verification of Stack-Space Bounds for C Programs. In *Conf. on Prog. Lang. Design and Impl. (PLDI'14)*, page 30, 2014.

[26] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional Certified Resource Bounds. In *36th Conf. on Prog. Lang. Design and Impl. (PLDI'15)*, 2015.

[27] P. Cerný, T. A. Henzinger, L. Kovács, A. Radhakrishna, and J. Zwirchmayr. Segment Abstraction for Worst-Case Execution Time Analysis. In *24th European Symposium on Programming (ESOP'15)*, 2015.

[28] A. Charguéraud and F. Pottier. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving - 6th International Conference (ITP'15)*, 2015.

[29] E. Çiçek, D. Garg, and U. A. Acar. Refinement Types for Incremental Computational Complexity. In *24th European Symposium on Programming (ESOP'15)*, 2015.

[30] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009.

[31] K. Crary and S. Weirich. Resource Bound Certification. In *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*, pages 184–198, 2000.

[32] N. A. Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th ACM Symp. on Principles Prog. Langs. (POPL'08)*, 2008.

[33] N. Danner, D. R. Licata, and R. Ramyaa. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

[34] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22nd USENIX Security Symposium*, Washington, D.C., 2013. USENIX.

[35] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, 2000.

[36] A. Flores-Montoya and R. Hähnle. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposiu (APLAS'14)*, 2014.

[37] F. Frohn, M. Naaf, J. Hensel, M. Brockschmidt, and J. Giesl. Lower Runtime Bounds for Integer Programs. In *Automated Reasoning - 8th International Joint Conference (IJCAR'16)*, 2016.

[38] J. Goguen and J. Meseguer. Security Policies and Security Models. In *In Proceedings of the 1982 IEEE Symposium on Security and Privacy*, 1982.

[39] B. Grobauer. Cost Recurrences for DML Programs. In *6th Int. Conf. on Funct. Prog. (ICFP'01)*, pages 253–264, 2001.

[40] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.

[41] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, 2009.

[42] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proceedings of the 20th Annual USENIX Security Symposium*, 2011.

[43] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. ISBN 9781107029576.

[44] D. Hedin and D. Sands. Timing aware information flow security for a javacard-like bytecode. *Electron. Notes Theor. Comput. Sci.*, 141(1), Dec. 2005.

[45] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *In Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, 1998.

[46] J. Hoffmann. RAML Web Site, 2016.

[47] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th Euro. Symp. on Prog. (ESOP'10)*, 2010.

[48] J. Hoffmann and Z. Shao. Type-Based Amortized Resource Analysis with Integers and Arrays. In *12th International Symposium on Functional and Logic Programming (FLOPS'14)*, 2014.

[49] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th ACM Symp. on Principles of Prog. Langs. (POPL'11)*, 2011.

[50] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012.

[51] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, 2003.

[52] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, 2006.

[53] M. Hofmann and G. Moser. Amortised Resource Analysis and Typed Polynomial Interpretations. In *Rewriting and Typed Lambda Calculi (RTA-TLCA;14)*, 2014.

[54] M. Hofmann and G. Moser. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, 2015.

[55] M. Hofmann and D. Rodriguez. Automatic Type Inference for Amortised Heap-Space Analysis. In *22nd Euro. Symp. on Prog. (ESOP'13)*, 2013.

[56] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th Symp. on Form. Meth. (FM'09)*, 2009.

[57] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, 2010.

[58] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *16th Annual International Cryptology Conference (CRYPTO'96)*, 1996.

[59] B. Köpf and D. Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[60] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th International Conference on Computer Aided Verification*, 2012.

[61] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Proceedings of the 22nd Annual IEEE Computer Security Foundations Symposium*, July 2009.

[62] U. D. Lago and M. Gaboardi. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, 2011.

[63] U. D. Lago and B. Petit. The Geometry of Types. In *40th ACM Symp. on Principles Prog. Langs. (POPL'13)*, 2013.

[64] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[65] P. Mardziel, M. S. Alvim, M. Hicks, and M. R. Clarkson. Quantifying information flow for dynamic secrets. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[66] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology*, 2006.

[67] T. Nipkow. Amortized Complexity Verified. In *Interactive Theorem Proving - 6th International Conference (ITP'15)*, 2015.

[68] L. Noschinski, F. Emmes, and J. Giesl. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Autom. Reasoning*, 51(1):27–56, 2013.

[69] O. Olivo, I. Dillig, and C. Lin. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Conference on Programming Language Design and Implementation (PLDI'15)*, 2015.

[70] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptography*, 2006.

[71] C. Pasareanu, Q.-S. Phan, and P. Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium*, 2016.

[72] F. Pottier and V. Simonet. Information Flow Inference for ML. In *In Proceedings of the 29th ACM Symposium on Principles Of Programming Languages*, 2002.

[73] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[74] A. Sabelfeld and A. C. Myers. Language-Based Information Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1): 5–19, 2003.

[75] H. R. Simões, P. B. Vasconcelos, M. Florido, S. Jost, and K. Hammond. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *17th Int. Conf. on Funct. Prog. (ICFP'12)*, 2012.

[76] V. Simonet. Flow Caml. `http://cristal.inria.fr/~simonet/soft/flowcaml/`, 2003.

[77] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*, 2014.

[78] M. Sulzmann, M. Müller, and C. Zenger. Hindley/Milner Style Type Systems in Constraint Form. In *Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science*, 1999.

[79] R. E. Tarjan. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods*, 6(2):306–318, 1985.

[80] P. Vasconcelos. *Space Cost Analysis Using Sized Types.* PhD thesis, School of Computer Science, University of St Andrews, 2008.

[81] P. B. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *Int. Workshop on Impl. of Funct. Langs. (IFL'03)*, 2003.

[82] P. B. Vasconcelos, S. Jost, M. Florido, and K. Hammond. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *24th European Symposium on Programming (ESOP'15)*, 2015.

[83] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. In *Journal of Computer Security*, pages 167–187, 1996.

[84] D. Walker. Substructural Type Systems. *Advanced Topics in Types and Programming Languages. MIT Press*, pages 3–43, 2002.

[85] R. Wilhelm et al. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[86] E. Yarrkov. Cryptanalysis of XXTEA, 2010. URL `http://eprint.iacr.org/2010/254`.

[87] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[88] D. Zhang, A. Askarov, and A. C. Myers. Language-Based Control and Mitigation of Timing Channels. In *In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[89] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, 2014.

[90] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symp. (SAS'11)*, 2011.