Type-Driven Verification of Non-functional Properties

Christopher Brown*
cmb21@st-andrews.ac.uk
University of St Andrews
Scotland, UK

Adam D. Barwell
adb23@st-andrews.ac.uk
University of St Andrews
Scotland, UK

Yoann Marquer
yoann.marquer@inria.fr
Inria, Univ Rennes, CNRS, IRISA
France

Céline Minh
celine.minh@inria.fr
Inria, Univ Rennes, CNRS, IRISA
France

Olivier Zendra
olivier.zendra@inria.fr
Inria, Univ Rennes, CNRS, IRISA
France

ABSTRACT

Energy, Time and Security (ETS) properties of programs are becoming increasingly prioritised by developers, especially where applications are running on ETS sensitive systems, such as embedded devices or the Internet of Things. Moreover, developers currently lack tools and language properties to allow them to reason about ETS. In this paper, we introduce a new *contract specification framework*, called *Drive*, which allows a developer to reason about ETS or other non-functional properties of their programs as *first-class* properties of the language. Furthermore, we introduce a contract specification language, allowing developers to reason about these first-class ETS properties by expressing contracts that are proved correct by an underlying formal type system. Finally, we show our contract framework over a number of representable examples, demonstrating provable worst-case ETS properties.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; *Imperative languages*; *Software maintenance tools*; • **Security and privacy →** *Software security engineering*.

KEYWORDS

IDRIS, C, time, energy, security, non-functional properties, proofs, verification, contracts

ACM Reference Format:
Christopher Brown, Adam D. Barwell, Yoann Marquer, Céline Minh, and Olivier Zendra. 2019. Type-Driven Verification of Non-functional Properties. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Non-functional properties, such as time, energy and security, are becoming critically important in areas ranging from small-scale sensors, through smart cyber-physical systems and mobile devices, to data centres, despite it being treated as a secondary concern. As embedded devices become more commonplace, increasing is the pressing demand for more time and energy optimisation in applications that execute on them, leading to longer battery lives and more efficient software. Moreover, systems are becoming increasingly vulnerable to timing and power leakage resulting in security problems, such as side-channel attacks, for example. However, dealing with time, energy and security properties at the language level is usually a black art, where developers require the use of specialised skills, low-level tools and techniques.

Therefore, there is a clear and pressing need to expose non-functional concerns to the systems designer and implementor in an accessible way. What we need is a way to turn energy consumption, execution time, security level and other important (non-functional) program properties into first-class citizens. This will allow (non-expert) programmers to understand and directly manipulate them. Consequently, this will eliminate both the complex, time-consuming and error-prone use of extrinsic tools as well as large-scale continuous testing for non-functional program behaviour from the software engineering process and, thus, make the latter significantly more lean, agile and productive. By exposing energy, time, security, etc., as first-class citizens and by making them easy to use and understand for application programmers, we will eliminate the need for dual-expertise: the application programmer will be able to manipulate and precisely reason about energy, time, security, etc. as normal program values, directly interacting with analytical and optimisation frameworks. They will ensure that solutions meet system requirements as needed while at the same time they are highly optimised for metrics without strict requirements.

In this paper we present a novel framework, called *Drive*, to allow the programmer to model non-functional properties of time, energy and security in their C source-code as *first-class citizens*; express resource contracts on time, energy and security usage, directly through source-level constructs; and verify these contracts *fully automatically* using the dependently typed programming language, Idris. We give explicit proofs of these correct properties, together with a denotational semantics of our contract system. In this way, we show our system constructs verified contracts. We illustrate the technique by considering a number of examples in C, as might be commonly found in real-time embedded systems, from the Beebs benchmarks suite.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Contributions

(1) We introduce a new model, exploiting dependent types, for expressing non-functional properties as provable contracts and assertions.
(2) We introduce a new *Embedded Domain Specific Language* (EDSL), called *CSL*, or Contract Specification Language, allowing programmers to annotate their programs with non-functional properties and contracts.
(3) We present a novel contract framework, called *Drive*, for modelling Energy, Time and Security (ETS) or other non-functional properties of programs as *first-class* properties of the language, building upon CSL the aforementioned model.
(4) We evaluate our contract system on a number of examples from the Beebs benchmark suite, demonstrating that our contract system allows the programmer to provide provable contracts of worst-case time properties of their programs. We also show in principle examples of energy and security properties, leaving these as a study for future work.

2 BACKGROUND

2.1 Dependent Types

Idris is a pure, dependently typed functional language, developed at the University of St Andrews. The syntax of Idris is similar to Haskell [23], and like Haskell, Idris supports algebraic data types with pattern matching, type classes, and do-notation. Unlike Haskell, Idris evaluates its terms eagerly, and has a richer type system, supporting full-spectrum dependent types, allowing the programmer to define types that may be predicated on values. By extension, this allows properties within the program to be expressed in a type and verified by the type-checking system [5].

Dependent types extend the idea of static type systems typically used in functional programming to allow types to become *first class citizens*, i.e. to be passed as values and manipulated as values in the programming space, and, as values, can be passed as arguments to functions and be returned by functions, just like any over value in the language [43]. The power of making types first class is that types can now *depend* upon values. Consider the following Idris type.

```
1  NatOrList : (b : Bool) -> Type
2  NatOrList False = Nat
3  NatOrList True = List Nat
```

This program defines a new type, `NatOrList` that, for some boolean value, `b`, returns a type that is either `Nat`, representing the natural numbers, or `List Nat` , representing a list of the natural numbers. Extending this idea, we can now define the following Idris function.

```
1  ZeroOrNil : (b : Bool) -> NatOrList b
2  ZeroOrNil False = Z
3  ZeroOrNil True = []
```

Here, `ZeroOrNil` is a function that takes as argument a boolean value, `b`. If `b` is `False`, `ZeroOrNat` returns `Z`, representing the zeroth element of the natural numbers in standard peano arithmetic, and the empty list, `[]`, otherwise. The return type of this function therefore depends on the value of its argument. If the argument is `True` the return type is a natural number; if it is `False`, it is a list type.

Allowing types to depend upon values results in more expressive types that can be sufficiently tightly constrained in order to express some property of one or more values; e.g. that a given list is ordered. In conjunction with the Curry-Howard correspondence, where a type represents a proposition whose inhabitants represent the proof of the proposition [43], logical and arithmetic formulae may be expressed and reasoned about as types in a manner that is consistent with some underlying constructive logic. This allows us, for a given context, to determine not only whether some logical expression holds, but also the reason (proof) *why* the expression holds. Moreover, for some expressions, it is possible to *automatically* determine such proofs, which we discuss further in Section 5.

2.2 Deriving Models

As *Drive* is a fully automated framework from C source code through to a type-verified contract produced by Idris, we require a mechanism to transform C source code into Idris data-types, representing our C space of properties and assertions. In order to parse C, we make use of, and extend, an already existing C99 parser that is implemented in Haskell, called *Language C*[1], which is a Haskell library providing static analyses and generation of C code. It features a complete C parser and pretty printer for the *full C99* standard, and a large set of C11 CLang/GNU extensions. For this paper, we have modified and extended Language-C, so that it parses CSL statements and assertions, generating Idris code as a result. This generated Idris is then used by the Idris contract system, an example of the generated output is given in Listing 2. This allows us to fully automate our Drive framework: providing an automatic compiler from (C99) CSL to an Idris contract.

2.3 Non-functional Properties

In this paper we consider three non-functional properties that we consider to be most commonplace: *energy*, *time* and *security*. In terms of time, we consider the worst-case execution time (*WCET*) obtained by executing the code with various underlying profiling tools such as the WCC compiler produced by the University of Hamburg [13]. Energy measurements are obtained by measuring the amount of energy in Joules (J) that is used by a complete processor package; i.e. we measure the total energy that is drawn by each hardware CPU socket. Energy results are obtained manually for the purposes of this paper, using Intel's Running Average Power Limit (RAPL) [11] system to collect this information, using model specific registers (MSR) to access the RAPL measurements. The MSRs provide accurate power readings for various systems components within each package e.g. the GPU and DIMMs. The same mechanism also allows energy consumption to be controlled/capped. We take the RAPL estimation of energy for the overall execution of the application. RAPL records power usage in terms of Watts (i.e. Joules per second). We calculate energy usage by computing the rate of change in power per unit of time using the formulae shown below:

[1] available on Hackage at http://hackage.haskell.org/package/language-c

$$Energy = Power \times Time$$
$$Joules = Watts \times Seconds$$

Security encompasses many aspects of interest. Communication properties like confidentiality/secrecy [40] or integrity [9] of the sent/received data, or even vulnerability to the most powerful attacker [15], can be captured by information-flow policies [4] or trace-based properties like non-interference [17] to prevent covert channels (unintentional transmission of sensitive information by using intended communication channels). A side-channel is a way of transmitting information (purposely or not) to another system out of the standard (intended) communication channels. Side-channel attacks aim at breaking cryptography by exploiting information that is revealed by an algorithm's physical execution. For instance, observing execution time [28] or power profile [27] may reveal secret information. We focus on these time- and power-related security properties in Section 4.2, to emphasize the relations between these three non-functional properties. Another computational property is the vulnerability to faults [52] injected during the execution in order to retrieve sensitive information from the alteration of the behavior of the program.

In this paper, we omit the details of obtaining such non-functional information, leaving it for future work to extend the *Drive* system with suitable linkage to automated tools (such as WCC [13]) to obtain the information automatically. However, our system infers the resource usage costs from a variety of underlying tools that are both dynamic and static, rather than performing static analysis on the source code itself. This gives the advantage of relying on real-world resource costs that are taken from actual measurements of time and energy from the execution profile of the application. Examples of such underlying tools are the AbsInt tool for instruction-level energy analysis [49], TimeWeaver for hybrid worst-case execution time analysis [25], and WCC for worst-case timing analysis of C source code and static energy models, such as those provided by Eder et al. [34].

3 ENERGY, TIME AND SECURITY CONTRACTS

Drive is a framework for forming contracts to support the inexperienced and possibly unskilled developer with the tools and techniques to reason about time, energy and security properties as first class citizens of the underlying language. Although *Drive* is, in theory, completely *language independent*, we demonstrate its effectiveness and generality on the C programming language, a language very commonly used for programming embedded systems. The workflow for using *Drive* is shown in Figure 1.

Starting with an original C source program, the programmer first annotates the source with annotations, by using a *Contract Specification Language*, a domain-specific language that extends C with special annotations and contract specification definitions (described in Section 4) describing non-functional properties of their source code and assertions (or contracts) around those properties. This annotated source code, with non-functional properties and assertions, is then passed through an ANSI C(99) parser (described in Section 2.2), which takes the annotated source code and produces a generated Idris model of the source code. This generated Idris model is a high-level abstract representation of the C program with its CSL annotations and assertions, which we pass to an underlying automated proof engine (also implemented in Idris) in order to verify that the programmer-defined certificate is valid or not. In addition to verifying the certificate, the CSL source code may be also passed to underlying tooling that may acquire further non-functional information about the execution of the application, such as timing, energy and security information. If the contracts are met, the result of the whole process is a certified C source program.

3.1 Motivating Example

Listing 1: Dijkstra C example, showing annotations in CSL

```
1  int time_spent, comparison_time;
2  ...
3  int dijkstra(int G[NUM_NODES][NUM_NODES], int chStart, int chEnd)
4  {
5  ...
6  int comparison_time, time_spent;
7  while (qcount() > 0)
8       {
9          QITEM *tmp = dequeue (&iNode, &iDist, &iPrev);
10         if(tmp != 0)
11            free(tmp);
12         __csl_time_worst_iter(&comparison_time);
13         for (i = 0; i < NUM_NODES; i++)
14         {
15            iCost = G[iNode][i];
16            if (iCost != 0)
17            {
18               if ((NONE == rgnNodes[i].iDist) ||
19                     (rgnNodes[i].iDist > (iCost + iDist)))
20               {
21                  rgnNodes[i].iDist = iDist + iCost;
22                  rgnNodes[i].iPrev = iNode;
23                  enqueue (i, iDist + iCost, iNode);
24               }
25            }
26         }
27      }
28 ..
29 }
30 ...
31 __csl_time_worst(&time_spent);
32 output[output_count] = dijkstra(G,0,NUM_NODES-1);
33 ...
34 __csl_assert(time_spent <= (EDGES * log (NUM_NODES)) *
       comparison_time);
35 ...
```

We consider the Dijkstra algorithm as implemented in the Beebs [37] benchmarks as a motivating example; an extract of the code is shown in Listing 1. In this example, the programmer requires a guarantee of the worst-case complexity of the algorithm, which is $O(e \log v)$, where e = *number of edges* and v = *number of vertices*. This is achieved by adding annotations to the source code calling various CSL functions. These CSL functions obtain various non-functional properties about the program's execution, such as worst and average case time, energy usage and even security information (illustrated in Section 4.2). This is information that is obtained from underlying profiling and modelling tools. When the program is executed, the CSL framework will automatically call the underlying tooling and assign the results of the profiling and/or analysis to the C variable that is passed as an argument to the CSL function.

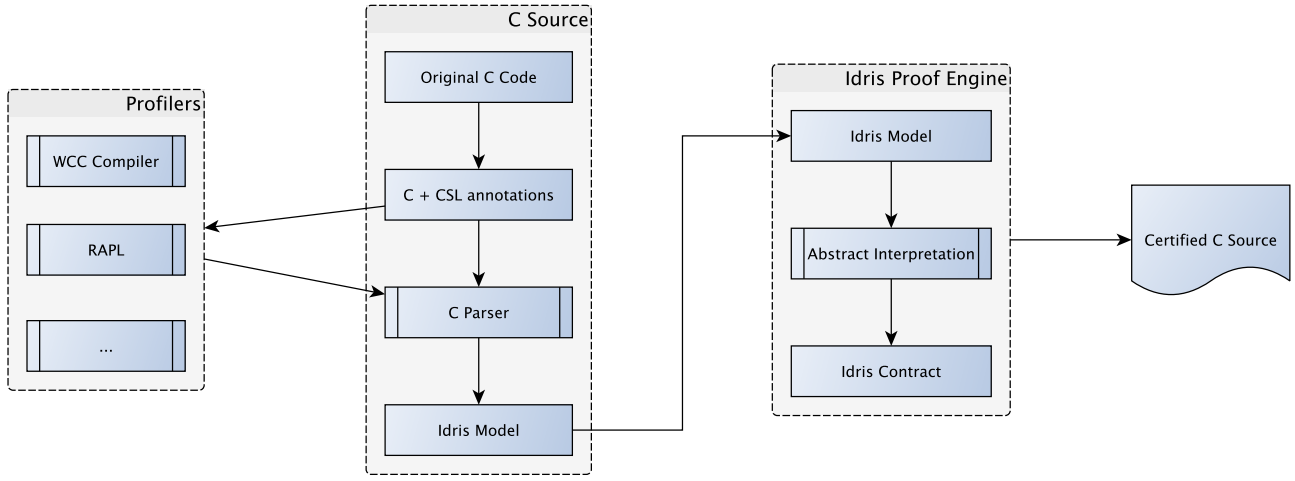


Figure 1: The *Drive* framework for producing contracts of non-functional properties of C programs

CSL annotations always immediately proceed a C statement (where a C statement can be any valid C statement) that is intended to be measured with either a time, energy or security non-functional property.

Examples of these CSL annotations are shown in Listing 1, where the programmer has added two annotations to the program source. The annotation on Line 12, `__csl_time_worst_iter(&comparison_time);` takes the average worst-case time metric for the loop defined immediately after, on Line 12, and assigns it to the regular C variable, `comparison_time`, which is defined, in the normal C fashion on Line 6; the average is calculated as the arithmetic mean over the loop iterations. The annotation on Line 31, `__csl_time_worst(&time_spent);`, takes the worst-case-execution-time (WCET) for the execution of the proceeding C statement on Line 32, and assigns it to the C variable, `time_spent` (defined on Line 6). This measurement is obtained by calling an underlying tool that calculates the worst case execution time and assigns the result to the C variable, `time_spent`. By assigning the values of the non-functional properties to regular C variables in this way allows the non-functional properties to become *first-class* properties of the C language itself: they can be used in the same way regular C variables can be used. This means they can be part of conditionals, loop bounds, etc.

Finally, the assertion of the complexity being an upper bound, is expressed as a contract in the program source. This contract is written as an assertion in the source code at Line 34 in Listing 1. This assertion is translated down into an underlying automated proof mechanism that generates a contract that the assertion holds for the given expression. In this paper, we use *Idris* as our proof mechanism, details of which are described in Section 5. In this example, the programmer states in the source code what the contract should be, by the statement, `__csl_assert(time_spent <= (EDGES * log (NUM_NODES) )* comparison_time);`. Here, `EDGES` and `NUM_NODES` are regular C variables that are defined in scope; `log` is the standard logarithmic function.

Listing 2: Example Idris code generated by C Parser for Dijkstra

```
1  import Drive
2  mutual
3         dijkstra : CLang
4         dijkstra = BlockTime "comparison_time" 23 $ DecVar "EDGES"
                1026 $ DecVar "NUM_NODES" 256 $ BlockTime "time_spent"
                32933 $ Assert dijkstra_assert $ Halt
5
6
7         dijkstra_assert : Env -> Assertion
8         dijkstra_assert env =
9             let
10                p0 = (Var "time_spent")
11                p0' = eval env p0
12
13                p1 = (Var "EDGES")
14                p1' = eval env p1
15
16                p2 = (Var "NUM_NODES")
17                p2' = eval env p2
18
19                p3 = (Var "comparison_time")
20                p3' = eval env p3
21
22                p4 = (Mul (Mul p1 (Log p2)) p3)
23                p4' = eval env p4
24            in
25                MkAssertion ( ( LTE p0 p4)
26                              (MkEvald p0 p0')
27                              (MkEvald p4 p4')
28                              (isLTE p0' p4' ))
```

The source code then passes through our *CSL compiler*: comprising an extended C parser, which parses the C and CSL constructs,

and generating an Idris program that is then executed by the contract system. The code that is generated by the parser is shown in Listing 2. In the listing, the Idris definition, `dijkstra`, on Line 5, represents an abstract high-level model of the C program in Listing 1, except that all of the superfluous structural detail have been removed, leaving only the information that is required to assert the contract. In this case, the only information that is required to verify the assertion are the variables representing the C variables of the assertion, and any variables being assigned values of non-functional information. `dijkstra_assert` models the assertion, where `p0` etc. are generated definitions. The types, `MkAssertion` etc. are described in more detail in Section 5.

Listing 3: Example Idris contract for Dijkstra showing verified assertion

```
1 ([("time_spent", 32933), ("NUM_NODES", 256), ("EDGES", 1026), ("
      comparison_time", 23)],
2  [MkAssertion (LTE (Var "time_spent")
3                    (Mul (Mul (Var "EDGES") (Log (Var "NUM_NODES"))) (
                          Var "comparison_time"))
4                    (MkEvald (Var "time_spent") 32933)
5                    (MkEvald (Mul (Mul (Var "EDGES") (Log (Var "
                          NUM_NODES"))) (Var "comparison_time"))
6                    ...
7                    (Yes (LTESucc (LTESucc ...
```

This generated Idris code is then passed to the Idris engine (described in Section 5), which produces a certificate, shown in Listing 3. In the listing, we see the result of verifying the assertion that was expressed in Listing 1 at Line 34 as a value in the `DecEq` type in Idris. Using `DecEq` gives us a decidable result as to whether the assertion holds for the given variables, or it does not. In Listing 3, we can see that, on Line 7, Idris returns `Yes`, indicating the assertion holds. This result is a certificate, proved correct by the underlying Idris type system.

4 CONTRACT SPECIFICATION LANGUAGE

The Contract Specification Language (CSL) is an *Embedded Domain Specific Language* (EDSL), allowing the programmer to annotate their source code in order to reason about non-functional properties as first-class citizens. CSL is both comprehensible to the programmer, and precise and detailed enough to support the required compiler optimisations and transformations. Therefore, CSL allows suitable annotations to be integrated into the program source, making it possible for the programmer and compiler to understand and reason about, e.g., energy, time, and security properties of the program. In order to illustrate the concept, we define CSL for C, but it can, in principle, be extended to other languages.

4.1 Expressing Time and Energy Properties

In order to express properties of time and energy, the programmer is required to annotate the source code with CSL annotations that make time and/or energy values available. Listing 4 shows the CSL annotations for assigning *time* values for a given statement, where annotations are defined as functions. These fall into three different aspects: worst-case (Line 1), best-case (Line 7), and average-case (Line 4) execution time. Each CSL annotation acts like a normal C function, where the parameter is a variable that is in scope. Each annotation must be placed before a regular C statement. Semantically, the annotation provides a measurement for the non-functional metric of the *proceeding* statement. Intuitively, this is equivalent to an assignment.

Listing 4: CSL time annotations

```
1 __csl_time_worst(&variable);
2 stmt1;
3
4 __csl_time_average(&variable);
5 stmt2;
6
7 __csl_time_best(&variable);
8 stmt3;
```

Listing 5: CSL time example

```
1 double loop_time;
2
3 __csl_time_worst(&loop_time);
4 for(int i =0; i < BOUND; i++){
5   f(i);
6 }
```

For example, in Listing 5, the `for`-loop (Lines 4–6) is annotated with `__csl_time_worst(&loop_time)` (Line 3) in order to obtain its worst-case execution time measurement. This value is stored in the variable, `loop_time`, which is passed by reference. The `__csl_time_worst` annotation will update the variable `loop_time` with the value of the metric measured.

Listing 6: CSL energy annotations

```
1 __csl_energy_worst(&variable);
2 stmt4;
3
4 __csl_energy_average(&variable);
5 stmt5;
6
7 __csl_energy_best(&variable);
8 stmt6;
```

Capturing energy measurements behaves in a very similar manner to capturing time measurements. Listing 6 shows the CSL annotations for assigning energy values. Similar to the time annotations of Listing 4, there are three possible annotations for energy: worst-case (Line 1), average-case (Line 4), and best-case (Line 7). These energy annotations also directly precede a C statement, capturing the measured energy measurement for the execution of that statement.

Listing 7: CSL example (revision 1)

```
1 double loop_time;
2
3 __csl_energy_worst(&loop_energy);
4 for(int i =0; i < BOUND; i++){
5   f(i);
6 }
```

For example, we might modify Listing 5 to measure energy in instead of time by replacing the annotation on Line 3. The resulting code, in Listing 7, will assign `loop_energy` the measurement for the worst-case energy usage for the `for`-loop.

Listing 8: CSL accumulator annotations

```
1  for(...) {
2    __csl_time_average_iter(&variable)
3    stmt1;
4
5    __csl_time_worst_iter(&variable)
6    stmt2;
7
8    __csl_time_best_iter(&variable);
9    stmt3;
10 }
```

CSL can be further extended with measurement annotations by using an underlying tool capable of making such desired measurements. One useful set of annotations, for example, is for the purpose of measuring statements within a loop context. Listing 8 shows the CSL annotations for assigning time values for a given statement that is within the body of a loop. As before, there are three kinds: worst-case (Line 4), best-case (Line 7), and average-case (Line 2). The worst-case annotation captures the worst time seen for `stmt2` over all iterations of the loop. Similarly, the best-case annotation captures the best time seen for `stmt3` over all iterations of the loop. Finally, the average-case annotation captures the mean average time of `stmt1` over all iterations of the loop. For example, in Listing 9, we extend Listing 5 by adding an accumulator annotation.

Listing 9: CSL accumulator example

```
1  double loop_time;
2  double ave_stmt_time;
3
4  __csl_time_worst(&loop_time);
5  for(int i =0; i < BOUND; i++){
6    __csl_time_average_iter(&ave_stmt_time);
7    f(i);
8  }
```

Here, the annotation on Line 5 captures the average amount of time the `f(i)` operation on Line 6 takes for $0 < i < \text{BOUND}$.

For all declaration annotations above, it is possible to change the default unit by supplying an optional string parameter that lets the programmer choose an appropriate unit to measure; e.g., in *milliseconds*, *seconds*, *nanoseconds*, for time, and *joules* and *watts·seconds* for energy. In our illustrative implementation, the default option for time is *milliseconds* and the default for energy is *joules*.

4.2 Expressing Security Properties

CSL annotations may be extended to non-functional properties other than time and energy. In order to illustrate this, we consider a small subset of security properties, with particular emphasis on side-channel attacks. We define three annotations in Listing 10 for capturing security levels for both side-channel attacks (Lines 1 and 4) and fault injection (Line 7). As in Section 4.1, security annotations act like a normal C function, where the parameter is a variable that is in scope. Each annotation is placed before a regular

Listing 10: CSL annotations for security properties

```
1  __csl_security_sca_time(&variable);
2  stmt1;
3
4  __csl_security_sca_power(&variable);
5  stmt2;
6
7  __csl_security_fault_injection(&variable);
8  stmt3;
```

C statement. Semantically, the annotation provides a *security level* for the proceeding statement. For the purposes of this paper, the security level of a statement is obtained from a given tool, e.g. [38] for time, that inspects how secure the code is for that particular property, and for a produced binary on a given architecture. Here, we assume that the security level is expressed as an integer.

In order to demonstrate these annotations/properties, we consider variants computing the (left-to-right) modular exponentiation by squaring, which is commonly used in, e.g., RSA [39]. They return the result of a^k *mod* n computed with an accumulator, x, where a and n are public variables, and k is some secret key.

Listing 11: Security Example: Square-and-Multiply

```
1  int sqmul(int a, unsigned int k, int n) {
2    int x = 1;
3    for (int i = 8*sizeof(int) - 1 ; i >= 0 ; i--) {
4      x = x*x % n;
5      if ((k >> i) & 1) {
6        x = x*a % n;
7      }
8    }
9    return x;
10 }
```

The condition of the `if`-statement on Line 5 detects the value of the i^{th} bit of the secret k, expressed in binary form. For every iteration, Listing 11 computes the multiplication only if that bit is a 1, which can be detected by observing execution time [28] or a power profile [27], and thus leads to information leakage from both time and power side-channel attacks.

Listing 12: Square-and-Multiply-Delay

```
1  int sqmul_delay(int a, unsigned int k, int n) {
2    int x = 1;
3    for (int i = 8*sizeof(int) - 1 ; i >= 0 ; i--) {
4      x = x*x % n;
5      if ((k >> i) & 1) {
6        __csl_time_average_iter(&time_mult)
7        x = x*a % n;
8      } else {
9        delay(time_mult);
10     }
11   }
12   return x;
13 }
```

Listing 12 is less vulnerable to time side-channel attacks because the imbalance in execution time of the `if`-statement branches has been addressed by introducing a delay in the else branch. `time_mult` is the average execution time for the multiplication operation in the then

Listing 13: Square-and-Multiply Comparison: Time SCA

```
1  int sec_lvl_time, sec_lvl_time_delay;
2
3  __csl_security_sca_time(&sec_lvl_time);
4  int x = sqmul(a, k, n);
5
6  __csl_security_sca_time(&sec_lvl_time_delay);
7  int y = sqmul_delay(a, k, n);
```

branch, obtained by using (Line 6) the time-capturing annotation `__csl_time_average_iter`.

The CSL annotation for time side-channel attacks from Listing 10 can be used to measure calls to both `sqmul` and `sqmul_delay`, as in Listing 13, where we would expect that `sqmul_delay` returns a higher security level than the call to `sqmul`. However, this is not necessarily the case for power profiles, that can be read (Simple Power Analysis [27]) in order to detect whether or not a multiplication has been computed during an iteration. In order to mitigate this, another variant of the square-and-multiply function might be employed.

Listing 14: Square-and-Multiply-Always

```
1  int sqmul_always(int a, unsigned int k, int n) {
2    int x = 1;
3    int y;
4    for (int i = 8*sizeof(int) - 1 ; i >= 0 ; i--) {
5      x = x*x % n;
6      if ((k >> i) & 1) {
7        x = x*a % n;
8      } else {
9        y = x*a % n;
10     }
11   }
12   return x;
13 }
```

Here, a dummy variable, `y`, is introduced and both branches of the `if` -statement perform the same operations. Similar to the time-based security level in Listing 13, security levels based on power profiles can be measured using the annotation in Listing 10.

Listing 15: Square-and-Multiply Comparison: Power SCA

```
1  int sec_lvl_power_delay, sec_lvl_power_always;
2
3  __csl_security_sca_power(&sec_lvl_power_delay);
4  int x = sqmul_delay(a, k, n);
5
6  __csl_security_sca_power(&sec_lvl_power_always);
7  int y = sqmul_always(a, k, n);
```

Here, security levels are stored in both `sec_lvl_power_delay` and `sec_lvl_power_always`, where we would normally expect the security level for `sqmul_always` to be greater than `sqmul_delay`. But the multiplication in the else branch of Listing 14 is a dummy operation, so injecting a fault [52] in `x*a % n` will change the final result only if the current secret bit is 1, thus leaking information. This is not the case with the Montgomery Ladder [24] where both variables x and y are interleaved accumulators. In Listing 16, a fault injected will propagate through successive iterations, so will alter the final result irrespective of the value of the current secret bit, preventing the opponent to obtain secret information. The security level of statements against fault-injection may be captured by the corresponding annotation in Listing 10.

Listing 16: Montgomery Ladder

```
1  int montgomery_ladder(int a, unsigned int k, int n)
2  {
3    int x = 1;
4    int y = a;
5    for (int i = 8*sizeof(int) - 1 ; i >= 0 ; i--) {
6      if ((k >> i) & 1) {
7        x = x*y % n;
8        y = y*y % n;
9      } else {
10       y = y*x % n;
11       x = x*x % n;
12     }
13   }
14   return x;
15 }
```

Listing 17: Square-and-Multiply Comparison: Fault-Injection

```
1  int sec_lvl_fault_always, sec_lvl_fault_ladder;
2
3  __csl_security_fault_injection(&sec_lvl_fault_always);
4  int x = sqmul_always(a, k, n);
5
6  __csl_security_fault_injection(&sec_lvl_fault_ladder);
7  int y = montgomery_ladder(a, k, n);
```

Similar to Listings 13 and 15, Listing 17 demonstrates the fault-injection-based annotation, capturing security levels for calls to `sqmul_always` and `montgomery_ladder`. Here, we would expect the security level of `montgomery_ladder` to be greater than `sqmul_always`.

4.3 Assertions

Assertions in CSL allow programmers to express contracts about their programs over the variables defined within it; e.g. the comparisons between security levels in Section 4.2. These variables may be used to capture non-functional properties of, e.g., time, energy, and security, as described in this section, or they may be normal variables that are in scope in the language for which CSL is defined. The assertion is checked using the model described in Section 5. An assertion takes the following form.

```
1  __csl_assert( bool_expression );
```

Here, `bool_expression` is an expression that returns a boolean value. The exact definition of well-formed expressions is derived from the underlying model since it is the model that determines whether the assertion holds, for a given context. In our exemplar implementation, this is a simple expression language comprising basic arithmetic operators and the binary operations: conjunction, disjunction, equality, and the standard inequalities over numbers. This may be extended to allow more complex expressions, given an extended model and/or parser (Section 2.2).

In order to demonstrate the use of assertions, we extend Listing 9.

Listing 18: CSL time example

```
1  double loop_time;
2  double iter_time;
3
4  __csl_time_worst(&loop_time);
5  for(int i =0; i < BOUND; i++){
6    __csl_time_average_iter(&iter_time)
7    f(i);
8  }
9
10 __csl_assert(loop_time <= (iter_time * BOUND));
```

Here, `loop_time <= (iter_time * BOUND)` is a normal boolean C expression. `loop_time`, `iter_time` and `BOUND` are all variables that are in scope and contain values at the point of the assertion. The programmer can call `__csl_assert` at any point in her C program, provided the free variables within the boolean expression are in scope and have values assigned to them. Assertions regarding security properties are expressed similarly. For example, given the square-and-multiply functions in Listings 11 and 12, we might extend Listing 13 with an assertion. Here, the security levels of the two function calls for time side-

```
1  int sec_lvl_time, sec_lvl_time_delay;
2
3  __csl_security_sca_time(&sec_lvl_time);
4  int x = sqmul(a, k, n);
5
6  __csl_security_sca_time(&sec_lvl_time_delay);
7  int y = sqmul_delay(a, k, n);
8
9  __csl_assert(sec_lvl_time <= sec_lvl_time_delay);
```

channel attacks are captured in `sec_lvl_time` and `sec_lvl_time_delay` by the annotations on Lines 4 and 7, respectively, and the assertion is the simple inequality, `sec_lvl_time <= sec_lvl_time_delay`.

4.4 Energy, Time and Security Properties as First-class Citizens

An advantage to our approach is that lifting these non-functional properties to the source level makes them *first-class citizens* of the source language. This means that if the programmer annotates their C source code with time, energy and security properties, these properties are assigned to standard C variables, and can be used in the usual way C variables can. For example, in the listing below, the programmer uses CSL to obtain the energy usage of a simple loop, and then following the loop, the energy usage is used in an if statement.

```
1  __csl_energy_worst(&loop_energy);
2  for ( ...) {
3   ...
4  }
5
6  if (loop_energy <= 20 )
7       securealgorithm1();
8  else
9       secureAlgorithm2();
```

Here, `loop_energy` is obtained by calling the CSL construct at Line 1 in the expected way, and then used, in Line 6, as part of an if statement. Here, `loop_energy` is checked to make sure that it does not exceed 20 joules of energy, executing different secure algorithms depending on the value of `loop_energy` as a result of the comparison.

5 CONTRACT DEFINITION AND DERIVATION

An annotated C program is modelled using Idris in order to determine whether assertions within the program hold. Models of programs are automatically derived using the parser from Section 2.2 and are inhabitants of the type `CLang`.

Listing 19: Idris model type

```
1  data CLang : Type where
2  -- CSL
3      BlockTime : (x : String) -> (v : Nat) -> (k : CLang) -> CLang
4      BlockEnergy : (x : String) -> (v : Nat) -> (k : CLang) -> CLang
5      Assert : (asn : Env -> Assertion) -> (k : CLang) -> CLang
6  -- C Structures
7      DecVar : (x : String) -> (v : Nat) -> (k : CLang) -> CLang
8      Halt   : CLang
```

`CLang` abstracts away all aspects of the annotated C program aside from CSL assertions and the assignments of variables used in those assertions; denoted respectively by `Assert`, `BlockTime`, `BlockEnergy` and `DecVar`. Variables, `x`, in models are unique and represent only the value that is live within the assertion. All values are modelled using natural numbers (`Nat`) to facilitate proof derivation; extending this to other types is a concern for future work. Where variables are not natural numbers in the original program, it is currently left to the parser (Section 2.2) to denote those original values consistently into natural numbers.

Models are in continuation passing style, where, as is standard, `k` denotes the continuation, and `Halt` denotes the end of a continuation. For example, the assertion `__csl_assert(x > 12);`, when `x = 13`, can be modelled as

```
1  DecVar "x" 13 (Assert asn Halt)
```

where `asn` is a function that models the assertion `x > 12`.

Assertions are boolean expressions that have a truth value, expressed via type inhabitance. Models can be constructed that contain assertions that *do not* hold. This allows a report to be presented to the programmer detailing whether, and *how*, each assertion holds. Since assertions are likely to have free variables, the model must specify a context, `Env`, via the function argument to the `Assert` constructor, that provides definitions for those variables (Listing 19, Line 5, `asn : Env -> Assertion`). Contexts are defined as a mapping of variable names to values; e.g. here, `Env = List (String,Nat)`. Proof terms of assertions are constructed by computing the normal forms of assertions; contexts provide definitions for free variables in assertions, allowing this. For example, in Listing 2, the variables in `dijkstra_assert` are substituted for their values by lookup in `env`, which is generated from definitions in the rest of the model, `dijkstra`. The type of assertions, `Assertion` (Listing 19, Line 5), is effectively a type alias for the type of boolean expressions, `BooleanExpression`. Boolean expressions comprise conjunction (`And`) and disjunction (`Or`) over boolean expressions, and equality (`Eq`) and ($\leq$) (`LTE`) operations

Listing 20: Assertion and boolean expressions definition

```
1  data Assertion : Type where
2     MkAssertion : BooleanExpression -> Assertion
3
4  data BooleanExpression : Type where
5     And : (x : BooleanExpression)
6        -> (y : BooleanExpression)
7        -> (xNF : BEvald x x')
8        -> (yNF : BEvald y y')
9        -> (prf : Dec (TyAnd x' y'))
10       -> BooleanExpression
11
12    Or  : (x : BooleanExpression)
13       -> (y : BooleanExpression)
14       -> (xNF : BEvald x x')
15       -> (yNF : BEvald y y')
16       -> (prf : Dec (TyOr x' y'))
17       -> BooleanExpression
18
19    Eq  : (x : NumericExpression)
20       -> (y : NumericExpression)
21       -> (xNF : Evald x x')
22       -> (yNF : Evald y y')
23       -> (prf : Dec (x' = y'))
24       -> BooleanExpression
25
26    LTE : (x : NumericExpression)
27       -> (y : NumericExpression)
28       -> (xNF : Evald x x')
29       -> (yNF : Evald y y')
30       -> (prf : Dec (x' `LTE` y'))
31       -> BooleanExpression
```

over natural numbers. Semantically, these functions are defined in the usual way. Our implementation includes the other standard inequality operators, but these are omitted here for clarity, since they can be defined in terms of ($\leq$).

Under our representation in Listing 20, each boolean expression is a binary operation, with arguments denoted `x` and `y`. Boolean expressions are extended with proof terms, `xNF` and `yNF`, for the relations `BEvald` and `Evald`, and the truth value for the expression, `prf`. Both `xNF` and `yNF` relate, depending on the operation, boolean or numeric expressions with their normal forms. We say `BEvald x x'`, where `x` is a boolean expression and `x'` is the boolean (`Bool`) to which `x` evaluates; and `Evald x x'`, where `x` is a numeric expression (Listing 21) and `x'` is the natural number (`Nat`) to which `x` evaluates. The truth term for the corresponding predicate, `prf`, is defined using the decidable proposition (`Dec`) and an underlying proposition that reflects the operation. For example, the truth value of `And` (Listing 20, Line 9) has the type `Dec (TyAnd x' y')`, where `TyAnd` is a type representing logical conjunction. When `And` evaluates to `True`,

```
1  data TyAnd : Bool -> Bool -> Type where
2     MkAnd : TyAnd True True
```

`prf = Yes prf'`, where `prf'` is a constructive proof that `TyAnd x' y'` is inhabited for the given `x'` and `y'`. For example, given `prf : TyAnd True True`, `prf = Yes MkAnd`). Similarly, when `And` evaluates to `False`, `prf = No prf'`, where `prf'` is proof that the underlying proposition, `TyAnd x' y'`, would be a contradiction (i.e. is uninhabited) for the given `x'` and `y'`.

Both equality and ($\leq$) operations range over numeric expressions, comprising literal values (`Lit`), variables (`Var`), addition (`Add`), subtraction (`Sub`), multiplication (`Mul`), truncated division (`Div`), the truncated logarithm to base 2 (`Log`), and the modulo operation (`Mod`).

Listing 21: Numeric expressions definition

```
1  data NumericExpression : Type where
2     Lit   : (n : Nat) -> NumericExpression
3     Var   : (x : String) -> NumericExpression
4     Log   : (x : NumericExpression) -> NumericExpression
5     Plus  : (x : NumericExpression) -> (y : NumericExpression) ->
           NumericExpression
6     Sub   : (x : NumericExpression) -> (y : NumericExpression) ->
           NumericExpression
7     Mul   : (x : NumericExpression) -> (y : NumericExpression) ->
           NumericExpression
8     Div   : (x : NumericExpression) -> (y : NumericExpression) ->
           NumericExpression
9     Mod   : (x : NumericExpression) -> (y : NumericExpression) ->
           NumericExpression
```

Semantically, these functions are defined in the usual way. Values of both `BooleanExpression` and `NumericExpression` are denoted into their corresponding Idris prelude functions via `beval` and `eval`, respectively.

Listing 22: Evaluation functions for boolean and numeric expressions

```
1  beval : (env : Env) -> (b : BooleanExpression) -> Bool
2  beval env (And x y xNF yNF (Yes prf)) = True
3  beval env (And x y xNF yNF (No contra)) = False
4  beval env (Or x y xNF yNF (Yes prf)) = True
5  beval env (Or x y xNF yNF (No contra)) = False
6  beval env (Eq x y xNF yNF (Yes prf)) = True
7  beval env (Eq x y xNF yNF (No contra)) = False
8  beval env (LTE x y xNF yNF (Yes prf)) = True
9  beval env (LTE x y xNF yNF (No contra)) = False
10
11 eval : (env : Env) -> (e : NumericExpression) -> Nat
12 eval env (Lit n) = n
13 eval env (Log n) = assert_total $ log2 (eval env n)
14 eval env (Var name) = case lookup name env of
15                           Just value => value
16                           Nothing   => 0
17 eval env (Plus x y) = (eval env x) `plus` (eval env y)
18 eval env (Sub x y) = (eval env x) `minus` (eval env y)
19 eval env (Mul x y) = (eval env x) `mult` (eval env y)
20 eval env (Div x y) = assert_total $ (eval env x) `div` (eval env y)
21 eval env (Mod x y) = assert_total $ (eval env x) `mod` (eval env y)
```

Boolean expressions are reduced to a boolean value (`beval`) by discriminating on `prf`. Numeric expressions are reduced to a natural number (`eval`) by denoting each `NumericExpression` operation into the respective Idris prelude function over natural numbers. Variables (`Var`, Lines 14–16) are looked up in the context, `env`, by the list prelude function `lookup`. The definition and generation of the Idris model require that all free variables in assertions are defined in the context. Consequently, the `Nothing` branch (Line 16) will, by definition, never be reached, but is included because lookup returns a `Maybe` value and eval returns a `Nat`. In the future, we intend to reflect this property in the type, obviating the need for `lookup`. Similarly, the arithmetic operations `Log`, `Div`, and `Mod` require `assert_total` to ensure evaluation to normal form, since the respective Idris prelude functions, `log`, `div`, and `mod`, are *partially defined*. We ensure that only models for which there are defined instances of `log`, `div`, and `mod` are generated.

In principle, it is possible to guarantee within the system that, for those cases, only arguments for which there are defined behaviours may be constructed, obviating the use of `assert_total`.

Reporting the Contract. For each model, the context and list of assertions are derived using the function `mkAssertions : CLang -> List Assertion`. `mkAssertions` is a fold over a model, whose step function collects assertions from `Assert` constructors, and variables and values from `BlockTime`, `BlockEnergy`, and `DecVar` constructors. Construction of the list of assertions requires a proof of whether each assertion holds, which is derived automatically via the reduction functions `beval` and `eval`, in conjunction with decision procedures for each boolean expression that is introduced via model generation. Decision procedures comprise: `isAnd` for conjunction; `isOr` for disjunction; `decEq` from the Idris prelude for equality; and `isLTE` from the Idris prelude for ($\leq$). In order to illustrate this process, we consider the model defined in Listing 23: This represents an

Listing 23: The Idris Abstract Model

```
dijkstra : CLang
dijkstra = BlockTime "comparison_time" 23
         $ DecVar "EDGES" 1026
         $ DecVar "NUM_NODES" 256
         $ BlockTime "time_spent" 32933
         $ Assert dijkstra_assert
         $ Halt

dijkstra_assert : Env -> Assertion
dijkstra_assert env =
  let
    p0 = (Var "time_spent")
    p0' = eval env p0

    p1 = (Var "EDGES")
    p1' = eval env p1

    p2 = (Var "NUM_NODES")
    p2' = eval env p2

    p3 = (Var "comparison_time")
    p3' = eval env p3

    p4 = (Mul (Mul p1 (Log p2)) p3)
    p4' = eval env p4
  in
    MkAssertion $ LTE p0 p4
                      (MkEvald p0 p0')
                      (MkEvald p4 p4')
                      (isLTE p0' p4')
```

abstract interpretation [10] of the original annotated C program in Listing 1; `mkAssertions` represents the abstract execution. Operationally, it folds over `dijkstra`, constructing the context: At an `Assert`

```
env : Env
env = [("comparison_time",23)
      ,("EDGES",1026)
      ,("NUM_NODES",256)
      ,("time_spent",32933)]
```

constructor, the step function passes the above context, `env`, to the assertion function, `dijkstra_assert`. Since Idris is strictly evaluated, this will cause the normalisation of `p0` and `p4`, producing `p0' = 32933` and `p4' = 188784`, respectively, via the `(eval env)` (partially applied) function. The `LTE` constructor requires a value of type `Dec (LTE p0' p4)`, which is provided by the evaluation of the term `(isLTE p0' p4')`, or `(isLTE 32933 188784)`, substituting the values of both `p0'` and `p4'`. In this example, the prelude function, `isLTE`, is used to provide a value that inhabits the type `Dec (LTE p0' p4)`. Under the Curry-Howard correspondence and Intensional Type Theories [44] upon which Idris is based, the type `Dec (LTE p0' p4)` is a proposition and the result of `(isLTE 32933 188784)` is a *proof* of that proposition, and thus a proof of the assertion itself. Accordingly, we say that the result of the abstract evaluation of `dijkstra` is a mechanical proof of the assertions therein, where Idris is used as a proof assistant.

6 EVALUATION

We provide our experimental system setup and describe our evaluation results.

6.1 Experiment Setup

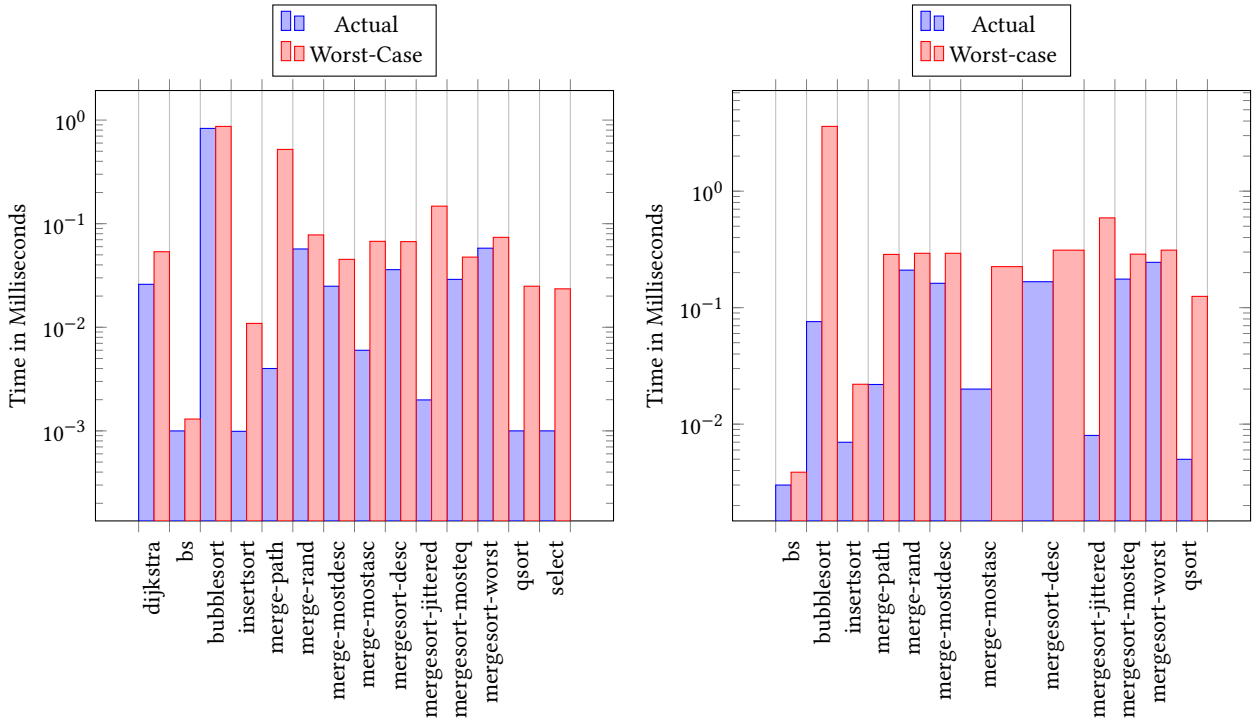
For our experiments we evaluate across two different machines. The first set of experiments, *corryvreckan*, are conducted on a server with Intel Xeon E5-2690 CPU with 28 cores, running at 2.6 Ghz with 256 GB of RAM, with the Scientific Linux 6.2 operating system with gcc 4.8.5 with optimisations turned off. The second set of experiments, *raspberrypi*, are executed on a Raspberry Pi Model 3 B, Quad-core 1.2Ghz Broadcom BCM2837 64bit CPU with 1 GB of RAM, with Raspbian GNU Linux (9) and gcc 6.3.0 with optimisations turned off.

6.2 The Beebs Benchmarks

For purposes of our evaluation, we use the Beebs benchmarks, which are designed for evaluating WCET and energy analysis for embedded systems [37]. In this paper, we chose a subset of the sorting algorithms from Beebs, where we measured the theoretical worse-case execution time based on the worst-case complexity of the sorting algorithm. For each sorting algorithm we annotated the source-code with CSL contructs measuring the average comparison time of the sorting algorithm, together with the total sorting time for the given input. The CSL assertion then asserts that the theoretical upper-bound (calculated from the average comparison time and the worst-case complexity) is indeed an upper bound for the given Beebs input data. In all cases, the contract system verifies that the worst-case is indeed an upper bound.

6.3 Experiments on *corryvreckan*

Our experiments for *corryvreckan* are summarised in Figure 3. In the figure, we show for each benchmark, the input size (i.e. the array size) of the given Beebs benchmark data, the calculated worst-case complexity of the algorithmn, the average operation (or comparison) time for the algorithm (measured using monotonic clock time, in milliseconds), the worst-case theoretical time, based on the worse-case complexity and the average operation time, the number of operations performed in the benchmark, and the total Beebs time. For each benchmark, we show that the worst-case time is an upper bound on the total runtime. In Figure 2(a), we show a summary of the *corryvreckan* results as a bar chart. Here, the y-axis is showing a logarithmic scale of milliseconds. The *blue* bar shows the actual time, and the *red* bar shows the theoretical worst-case time based



(a) Comparison of Beebs runtimes versus worst-case on *corryvreckan*

(b) Comparison of Beebs runtimes versus worst-case on *raspberrypi*

Figure 2: Comparison of Beebs runtimes versus worst-case theoretical bounds

on the average operation time and the worst-case complexity of the algorithm; in all cases the actual time is below the theoretical worst-case time. In all cases, the actual time does not exceed the worst-case time. For *mergesort-asc* and *mergesort-equal*, there were 0 operations performed due to the input lists already being sorted; we omit their results here.

6.4 Experiments on *raspberrypi*

Our experiments for the *raspberry pi* proceed in a similar manner to those shown for *corryvreckan*, and are given in Figure 4. The Raspberry Pi is a small embedded device, no untypical of the types of devices found in embedded systems. It allows us to model the benchmarks in an embedded system, in contrast to the multicore system, *corryvreckan*, and show the applicability of the *Drive* system to an embedded space. The benchmarks were executed in a similar manner to those for *corryvreckan*, with the same data inputs as supplied by the Beebs benchmarks. Again, *mergesort-asc* and *mergesort-desc* have 0 operations, so complete in almost 0 time; we again omit their results. We omit details of *dijsktra* and *select* for the Raspberry Pi due to segmentation faults in the executions. A comparison of the Beebs actual times (shown in *blue*) versus the theoretical worst-case time (shown in *red*) are given in Figure 2(b).

7 RELATED WORK

There is a great deal of work within the programming language community on the calculation and bounding of resource usage, generally with a focus on memory usage or worst-case execution time [8, 50]. Programming language approaches typically seek to allow the programmer to reason about resources at the type level, *inter alia*. Approaches include: the use of abstract interpretation to infer symbolic bounds [19]; the use of amortised analyses to automatically infer polynomial multivariate bounds [21]; the use of refinement types to ensure that successive iterations cost less than previous iterations in incremental computing [8]; the use of linear dependent types to bound the number of reductions that may be applied [30]; and, similarly, the use of sized types to analyse space usage [7, 22, 47]. Where these approaches generally operate within specific type systems or languages, applying them to other languages, e.g. C, would typically be a substantial task. Whilst a model of some program in the given target language could be constructed, these analyses would result in an analysis of *the model*, not the original program. Moreover, as static approaches, these do not necessarily reflect actual execution times since architectural aspects (e.g. caching) are not modelled within the (type) system. Conversely, the approach presented here uses concrete resource information derived from the original program, thereby allowing us to reason about the original program itself and in a manner that directly and accurately reflects resource usage.

Beyond time and space, recent work has exposed the energy consumption of a program in the form of a function that is parameterised by program arguments [16, 32]. Security aspects, such as information flow and leakage, have also been modelled using type

BEEBs Example	BEEBs Input Size	Worst-case compl.	Avg. Op. Time	# BEEBs Ops	Worst-case time	BEEBs time
dijkstra	10	$(v^2) + (e + v) \log v$	0.000227	220	**0.0536**	**0.02599**
bs	15	$\log n$	0.000333	3	**0.00130099**	**0.000999**
bubblesort	100	n^2	0.0000869	5050	**0.86**	**0.831**
insertsort	11	n^2	0.00009	9	**0.0109**	**0.00099**
mergesort-pathological	100	$n \log n$	0.0000784	51	**0.521**	**0.004**
mergesort-random	100	$n \log n$	0.00011728	486	**0.0779**	**0.057**
mergesort-mostdesc	100	$n \log n$	0.00006811989	367	**0.045257**	**0.0249**
mergesort-mostasc	100	$n \log n$	0.0001016949	59	**0.06756**	**0.006**
mergesort-desc	100	$n \log n$	0.000101124	356	**0.0671**	**0.036**
mergesort-jittered	100	$n \log n$	0.00022	9	**0.1476**	**0.00199**
mergesort-mosteq	100	$n \log n$	0.0000716	405	**0.04757**	**0.029**
mergesort-worst	100	$n \log n$	0.000111	522	**0.07382**	**0.058**
qsort	20	n^2	0.0000625	16	**0.0249**	**0.001**
select	20	n^2	0.00005882	34	**0.0235**	**0.001**

Figure 3: Experiments on *corryvreckan*. Times shown in *milliseconds*.

BEEBs Example	BEEBs Input Size	Worst-case compl.	Avg. Op. Time	# BEEBs Ops	Worst-case time	BEEBs time
bs	15	$\log n$	0.00099	3	**0.00386782**	**0.0030**
bubblesort	100	n^2	0.0003599	5050	**3.5999**	**0.0759**
insertsort	11	n^2	0.00018	9	**0.022**	**0.00699**
mergesort-pathological	100	$n \log n$	0.00043137	51	**0.28659**	**0.0219**
mergesort-random	100	$n \log n$	0.000434	486	**0.293**	**0.21**
mergesort-mostdesc	100	$n \log n$	0.0004	367	**0.293**	**0.1619**
mergesort-mostasc	100	$n \log n$	0.00033	59	**0.225**	**0.02**
mergesort-desc	100	$n \log n$	0.000469	356	**0.31166**	**0.167**
mergesort-jittered	100	$n \log n$	0.00088	9	**0.59**	**0.008**
mergesort-mosteq	100	$n \log n$	0.000434	405	**0.288**	**0.1759**
mergesort-worst	100	$n \log n$	0.000469	522	**0.31182**	**0.24499**
qsort	20	n^2	0.000312	16	**0.125**	**0.00499**

Figure 4: Experiments on *raspberry pi*. Times shown in *milliseconds*

systems [6, 48]. As additional non-functional properties become of interest, it becomes increasingly relevant to combine or compose said properties when reasoning about them in the program. To the best of our knowledge, existing approaches are generally limited to a single resource. The few approaches that do trade energy requirements with performance constraints [3, 51], with one notable exception [33], treat both properties as a constraint within the compilation/optimisation stage, and do not allow the programmer to reason about them. Conversely, and since the approach presented here builds upon, and uses, existing tools, it is both possible and simple to express assertions about multiple non-functional properties at the program level.

The aforementioned notable exception, the Ciao Preprocessor system (CiaoPP) [20, 33, 41], enables the programmer to infer resource usage, *and* reason about the correctness of programs with respect to resource usage information. CiaoPP achieves this by translating annotated programs in high-level languages, e.g. Java [18] and XC [1], into sequences of Horn clauses. This Horn Clause Intermediate Representation (HC IR) is used to reason about (the correctness of) transformations over the global state of the program. Despite the apparent similarities of both *Drive* and CiaoPP systems, beyond the choice of model representation, there are three distinct differences:

(1) the approach to resource analysis;
(2) the exposure of resource usage information to the programmer; and,
(3) the scope of the respective models.

CiaoPP uses its HC IR to derive resource usage information: reporting static resource analyses for both time and energy [33], and memory and the transmission of bits [41]. To the best of our knowledge, no security analyses have thus far been a subject of CiaoPP resource usage analysis. The use of external tooling in *Drive* represents a trade-off between the tightness of integration and assurances, as seen in CiaoPP, and an increased flexibility in choice of approaches to resource analyses. External tooling need not be limited to the use of a specific representation, but different tools may instead use their own internal representation, potentially allowing a wider range of analyses. However, external tooling must generally be trusted by the programmer, where trust might be derived from, i.e., empirical testing or external proofs.

The resource analyses available within the CiaoPP system return *functions* that take the size of relevant data as an argument. Conversely, the *Drive* system expects *ground values* for *specific instances* of the annotated statement. Moreover, unlike *Drive*, CiaoPP does not reflect resource usage values into the original program and thus this information *cannot be used in a first-class way*. Whilst it should be noted that the underlying Ciao does facilitate first-class resource usage values, CiaoPP does not propagate those values back into the original language [33].

Beyond resource analysis, the scope and approach to verification represents a significant difference between the two approaches. CiaoPP models the (concrete and intended) semantics of a given program written in a annotated high-level language using the HC IR, in part to facilitate its resource analyses. Conversely, *Drive* models only those parts of the program that are necessary for determining validity of CSL assertions; in part, facilitated by the use of external tools. The certificates derived from these models represent proofs by reflection or contradiction of whether an assertion holds. *Drive* can thus be seen as a proof-carrying approach [35], where the certificates are checkable by the Idris type checker. CiaoPP similarly provides the fixpoint computed by the preprocessor [2]. Due to the nature of the CiaoPP analysis, these fixpoints may become large as the program itself grows larger. Whilst the certificates produced by the *Drive* system are focussed on assertions and carry minimal information, due to the constructive nature of proofs in Idris, and the use of natural numbers in *Drive*, large numbers may produce large certificates. Future work will focus on making these proofs smaller, easier to read and understand, and investigate making these *proofs first-class*; i.e. accessible within the program being analysed. One consequence of the relative immaturity of the technique, is that the assertion language supported by CiaoPP is more expressive than that which is supported by *Drive*. Future work will consider increasing the expressiveness of the *Drive* assertion language; for example, allowing the inference of intervals for free variables in non-ground assertions.

Measurement and Inference of Resource Usage. The approach presented here is dependent upon other tools, e.g. the WCET-aware C Compiler [14], in order to provide resource usage values to the model. Unsurprisingly, the worst-case execution time (WCET) community has been a significant driver of such tools; a general overview of methods and tools is provided by Wilhelm *et al.* [50]. For example, one of the more popular methods used for WCET analysis is the Implicit Path Enumeration Technique, introduced by Li *et al.* [31], which estimates best- and worst-case times for embedded software without needing to enumerate all possible control-paths in a program.

Energy usage is of increasing interest due to the prevalence of battery-powered systems. The measurement of energy usage typically comes in three forms: 1) instruction set models for simple, usually predictable, embedded architectures [26, 46]; 2) hardware performance counter models that utilise more complex architectures' performance events (e.g. cache misses and branch miss-predictions) [42]; and 3) the use of linear regression on an amount of empirical data to extract unknown terms that can be utilised to construct an energy model [36].

Regarding security properties, in [12] the authors used Haskell's type system to enforce secure information flow (non-interference [17]), augmented with flexible declassification policies, and secure computation on untainted data (both verified if the source code compiles), and mandatory user input validation (verified at runtime). Their lightweight Haskell library for writing and reading files ensures that values in sensitive applications come from a trusted source of input, and provides a simple way for formalizing declassification policies. But the programmer is forced to validate every untrusted value and to provide two different paths (whether an error occurs or not) in the source code, while in our paper we are making no assumption on the developer behaviour.

FABLE [45] is a core formalism for a programming language in which programmers can 1) define custom security labels and associate them with the data they protect using dependent types, and 2) define the interpretation of labels in special enforcement policy functions separated from the rest of the program. This allows to write programs that enforce a variety of security policies, not restricted as above to information flow security policies, such as access control, static and dynamic information flow with forms of declassification, provenance tracking and policies based on security automata. FABLE does not guarantee that a security policy is correctly implemented, but its design simplifies proof of this fact.

As shown by both frameworks, in the literature the focus is on communication security properties like confidentiality/secrecy [40] or integrity [9] of the data. The security examples, properties and contracts expressed in this paper are more focused on preventing

information leakage from the computation itself (side-channel attacks and fault injection). Our approach is more general because it does not rely on particular primitives to express security properties.

Another general approach is LOCKS [29], which is a generic and formal DSL (domain specific language) that allows a security practitioner to express and compose (both qualitative and quantitative) security goals (embracing a wide number of attributes such as cost, damage, probability, etc.) in a declarative manner. These goals are expressed as queries over an attack model, the SAM (structural attack model), which is based on partially ordered sets of successful attacks that dictates which steps needs to be carried out and in which order, and is used as the semantic model for LOCKS. These security goals can be compared to the (more accessible) regular C expressions that we used for the assertions, and thus LOCKS can be compared to *Drive*. But they lack an automated tool like Idris to verify whether these goals are achieved or not.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel framework, called *Drive*, allowing the programmer to reason about Energy, Time and Security (ETS) properties of their source-code. Moreover, Drive allows the programmer to express these ETS properties as *first-class* properties of the underlying language. We also introduced a new DSL, called the Contract Specification Language (CSL) to allow programmers to annotate the C source-code with ETS properties and source-level provable contracts. These source-level contracts are verified by an underlying type system that models the high-level structures of the source-code, giving a decidable proof that the contract has been met. This is used to allow the programmer to, for example, place upper bounds on sections of their source code in terms of e.g. time and energy, or define a security level that they require to be met. We evaluated our approach using a set of sorting algorithms from the Beebs benchmarks suite, on a 28-core multi-core machine and a raspberry pi model 3 B. Both results allowed us to evaluate our framework on showing that the worst-case complexity of the sorting algorithms is indeed an upper-bound for the supplied Beebs benchmarks data sets.

In the future, we plan to extend our work in a number of new directions.

- We intend to make contexts parameters of the assertion type, define models such that variables carrying proofs of their inclusion in the context, and avoid the use of partial definitions for arithmetic operations. This will obviate the need for evaluation relations and an explicit lookup operation.
- We will extend the assertions to be able to take into account open terms. For example, the contract may be verified correct depending upon some assumption.
- We plan to extend our type-system to model the source code using a small-step semantics; this would enable us to model and verify assertions based on user-defined functions.
- We will evaluate our technique on a number of examples that are dependent on the energy consumption of the application w.r.t to the total energy budget of the device. For example, modelling the Beebs benchmarks on a small embedded device, such as the Discovery STM32.
- We plan to extend our framework on security contracts, including evaluating over a number of security benchmarks, testing security attacks such as side-channel attacks, etc.

ACKNOWLEDGEMENTS

We thank Annelie Heuser and Tania Richmond for their help regarding side-channel analysis and the Montgomery ladder.

This work was supported by the EU Horizon 2020 project, *TeamPlay* (https://www.teamplay-h2020.eu), grant number 779882, and UK EPSRC *Discovery*, grant number EP/P020631/1.

REFERENCES

[1] Xc specification ver. 1.0 (x5965a) (2011), https://www.xmos.com/developer/xc-specification
[2] Albert, E., Arenas, P., Puebla, G., Hermenegildo, M.V.: Certificate size reduction in abstraction-carrying code. TPLP **12**(3), 283–318 (2012)
[3] Andrei, A., Eles, P., Jovanovic, O., Schmitz, M.T., Ogniewski, J., Peng, Z.: Quasi-static voltage scaling for energy minimization with time constraints. IEEE Trans. VLSI Syst. **19**(1), 10–23 (2011)
[4] Bell, D.E., La Padula, L.J.: Secure computer system: Unified exposition and multics interpretation. In: Tech report ESD-TR-75-306. pp. 1–133. Mitre Corp, Bedford, Ma. (1976)
[5] Brady, E.C.: Idris —: Systems programming meets full dependent types. In: Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification. pp. 43–54. PLPV '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1929529.1929536, http://doi.acm.org/10.1145/1929529.1929536
[6] Chen, H., Tiu, A., Xu, Z., Liu, Y.: A permission-dependent type system for secure information flow analysis. In: CSF. pp. 218–232. IEEE Computer Society (2018)
[7] Chin, W., Khoo, S.: Calculating sized types. Higher-Order and Symbolic Computation **14**(2-3), 261–300 (2001). https://doi.org/10.1023/A:1012996816178, https://doi.org/10.1023/A:1012996816178
[8] Çiçek, E., Garg, D., Acar, U.A.: Refinement types for incremental computational complexity. In: Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. pp. 406–431 (2015). https://doi.org/10.1007/978-3-662-46669-8_17, https://doi.org/10.1007/978-3-662-46669-8_17
[9] Clark, D.D., Wilson, D.R.: A comparison of commercial and military computer security policies. In: 1987 IEEE Symposium on Security and Privacy. pp. 184–184 (April 1987). https://doi.org/10.1109/SP.1987.10001
[10] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252 (1977). https://doi.org/10.1145/512950.512973, https://doi.org/10.1145/512950.512973
[11] David, H., Gorbatov, E., Hanebutte, U.R., Khanna, R., Le, C.: Rapl: Memory power estimation and capping. In: Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design. pp. 189–194. ISLPED '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1840845.1840883, http://doi.acm.org/10.1145/1840845.1840883
[12] Di Pirro, M., Conti, M., Lazzeretti, R.: Ensuring information security by using haskell's advanced type system. In: 2017 International Carnahan Conference on Security Technology (ICCST). pp. 1–6 (Oct 2017). https://doi.org/10.1109/CCST.2017.8167844
[13] Falk, H., Lokuciejewski, P.: A compiler framework for the reduction of worst-case execution times. Real-Time Systems **46**(2), 251–300 (Oct 2010). https://doi.org/10.1007/s11241-010-9101-x, https://doi.org/10.1007/s11241-010-9101-x
[14] Falk, H., Lokuciejewski, P.: A compiler framework for the reduction of worst-case execution times. Real-Time Systems **46**(2), 251–300 (2010)
[15] Focardi, R., Martinelli, F.: A uniform approach for the definition of security properties. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM'99 — Formal Methods. pp. 794–813. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
[16] Georgiou, K., Kerrison, S., Chamski, Z., Eder, K.: Energy transparency for deeply embedded programs. TACO **14**(1), 8:1–8:26 (2017)
[17] Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy. pp. 11–11 (April 1982). https://doi.org/10.1109/SP.1982.10014
[18] Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 8 Edition. Addison-Wesley Professional, 1st edn. (2014)

[19] Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. pp. 127–139 (2009). https://doi.org/10.1145/1480881.1480898, https://doi.org/10.1145/1480881.1480898

[20] Hermenegildo, M.V., Puebla, G., Bueno, F., López-García, P.: Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). Sci. Comput. Program. **58**(1-2), 115–140 (2005). https://doi.org/10.1016/j.scico.2005.02.006, https://doi.org/10.1016/j.scico.2005.02.006

[21] Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. **34**(3), 14:1–14:62 (2012). https://doi.org/10.1145/2362389.2362393, https://doi.org/10.1145/2362389.2362393

[22] Hughes, J., Pareto, L.: Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In: Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999. pp. 70–81 (1999). https://doi.org/10.1145/317636.317785, https://doi.org/10.1145/317636.317785

[23] Hutton, G.: Programming in Haskell. Cambridge University Press, New York, NY, USA (2007)

[24] Joye, M., Yen, S.M.: The montgomery powering ladder. In: Kaliski, B.S., Koç, ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2002. pp. 291–302. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)

[25] Kästner, D., Pister, M., Wegener, S., Ferdinand, C.: TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis. In: Altmeyer, S. (ed.) 19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019). OpenAccess Series in Informatics (OASIcs), vol. 72, pp. 1:1–1:11. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). https://doi.org/10.4230/OASIcs.WCET.2019.1, http://drops.dagstuhl.de/opus/volltexte/2019/10766

[26] Kerrison, S., Eder, K.: Energy modeling of software for a hardware multithreaded embedded microprocessor. ACM Trans. Embedded Comput. Syst. **14**(3), 56:1–56:25 (2015)

[27] Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) Advances in Cryptology — CRYPTO' 99. pp. 388–397. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)

[28] Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Koblitz, N. (ed.) Advances in Cryptology — CRYPTO '96. pp. 104–113. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)

[29] Kumar, R., Rensink, A., Stoelinga, M.: Locks: A property specification language for security goals. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 1907–1915. SAC '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3167132.3167336, http://doi.acm.org/10.1145/3167132.3167336

[30] Lago, U.D., Petit, B.: The geometry of types. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 167–178 (2013). https://doi.org/10.1145/2429069.2429090, https://doi.org/10.1145/2429069.2429090

[31] Li, Y.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. IEEE Trans. on CAD of Integrated Circuits and Systems **16**(12), 1477–1487 (1997)

[32] Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., López-García, P., Grech, N., Hermenegildo, M.V., Eder, K.: Energy consumption analysis of programs based on XMOS isa-level models. In: LOPSTR. Lecture Notes in Computer Science, vol. 8901, pp. 72–90. Springer (2013)

[33] López-García, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F., Hermenegildo, M.V.: Interval-based resource usage verification by translation into horn clauses and an application to energy consumption. TPLP **18**(2), 167–223 (2018). https://doi.org/10.1017/S1471068418000042, https://doi.org/10.1017/S1471068418000042

[34] Morse, J., Kerrison, S., Eder, K.: On the limitations of analysing worst-case dynamic energy of processing. ACM Transactions on Embedded Computing Systems **17**(3) (2 2018). https://doi.org/10.1145/3173042

[35] Necula, G.C.: Proof-carrying code. In: Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997. pp. 106–119 (1997). https://doi.org/10.1145/263699.263712, https://doi.org/10.1145/263699.263712

[36] Núñez-Yáñez, J.L., Lore, G.: Enabling accurate modeling of power and energy consumption in an arm-based system-on-chip. Microprocessors and Microsystems - Embedded Hardware Design **37**(3), 319–332 (2013)

[37] Pallister, J., Hollis, S.J., Bennett, J.: BEEBS: open benchmarks for energy measurements on embedded platforms. CoRR **abs/1308.5174** (2013), http://arxiv.org/abs/1308.5174

[38] Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? In: Design, Automation Test in Europe Conference Exhibition (DATE), 2017. pp. 1697–1702 (March 2017). https://doi.org/10.23919/DATE.2017.7927267

[39] Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21**(2), 120–126 (1978)

[40] Ryan, P.Y.: Mathematical models of computer security. In: International School on Foundations of Security Analysis and Design. pp. 1–62. Springer (2000)

[41] Serrano, A., López-García, P., Hermenegildo, M.V.: Resource usage analysis of logic programs via abstract interpretation using sized types. TPLP **14**(4-5), 739–754 (2014). https://doi.org/10.1017/S147106841400057X, https://doi.org/10.1017/S147106841400057X

[42] Shao, Y.S., Brooks, D.M.: Energy characterization and instruction-level energy model of intel's xeon phi processor. In: ISLPED. pp. 389–394. IEEE (2013)

[43] Slama, F.: Automatic generation of proof terms in dependently typed programming languages. Ph.D. thesis, University of St Andrews (2018)

[44] Slama, F., Brady, E.: Automatically proving equivalence by type-safe reflection. In: Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings. pp. 40–55 (2017). https://doi.org/10.1007/978-3-319-62075-6_4, https://doi.org/10.1007/978-3-319-62075-6_4

[45] Swamy, N., Corcoran, B.J., Hicks, M.: Fable: A language for enforcing user-defined security policies. In: 2008 IEEE Symposium on Security and Privacy (sp 2008). pp. 369–383 (May 2008). https://doi.org/10.1109/SP.2008.29

[46] Tiwari, V., Malik, S., Wolfe, A.: Power analysis of embedded software: a first step towards software power minimization. IEEE Trans. VLSI Syst. **2**(4), 437–445 (1994)

[47] Vasconcelos, P.B., Hammond, K.: Inferring cost equations for recursive, polymorphic and higher-order functional programs. In: Implementation of Functional Languages, 15th International Workshop, IFL 2003, Edinburgh, UK, September 8-11, 2003, Revised Papers. pp. 86–101 (2003). https://doi.org/10.1007/978-3-540-27861-0_6, https://doi.org/10.1007/978-3-540-27861-0_6

[48] Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security **4**(2/3), 167–188 (1996)

[49] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 36:1–36:53 (May 2008). https://doi.org/10.1145/1347375.1347389, http://doi.acm.org/10.1145/1347375.1347389

[50] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools. ACM Trans. Embed. Comput. Syst. **7**(3), 36:1–36:53 (May 2008). https://doi.org/10.1145/1347375.1347389, http://doi.acm.org/10.1145/1347375.1347389

[51] Xian, C., Lu, Y., Li, Z.: Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time. In: DAC. pp. 664–669. IEEE (2007)

[52] Ziade, H., Ayoubi, R., Velazco, R.: A survey on fault injection techniques. International Arab Journal of Information Technology **Vol. 1, No. 2, July**, 171–186 (2004), https://hal.archives-ouvertes.fr/hal-00105562