Semi-automatic ladderisation: improving code security through rewriting and dependent types

Christopher Brown, Adam Barwell, Yoann Marquer, Olivier Zendra, Tania Richmond, Chen Gu

Semi-automatic Ladderisation: Improving Code Security through Rewriting and Dependent Types

Christopher Brown [cmb21@st-andrews.ac.uk] [1],
Adam D. Barwell [a.barwell@imperial.ac.uk] [1,2],
Yoann Marquer [yoann.marquer@inria.fr] [3],
Olivier Zendra [Olivier.Zendra@inria.fr] [3],
Tania Richmond [tania.richmond.nc@gmail.com] [3,4], and
Chen Gu [guchen@hfut.edu.cn] [5]

[1]University of St Andrews, Scotland, UK
[2]Imperial College London, UK
[3]Inria, Univ. Rennes, CNRS, IRISA, France
[4]DGA - Maîtrise de l'Information, Bruz, France
[5]Hefei University of Technology, Hefei, China

Keywords: Security, Soundness, Refactoring, Rewriting, Semantics, Side-channel, Fault injection, Dependent Types

Author version of the article published at PEPM 2022

Abstract

Cyber attacks become more and more prevalent every day. One type of cyber attack is known as a *side channel attack*, where attackers exploit information leakage from the physical execution of a program, e.g. timing or power leakage, to uncover secret information, such as encryption keys or other sensitive data. There have been various attempts at addressing the problem of preventing side-channel attacks, often relying on various measures to decrease the discernibility of several code variants or code paths. Most techniques require a high-degree of expertise by the developer, who often employs ad hoc, hand-crafted code-patching in an attempt to make it more secure. In this paper, we take a different approach: building on the idea of *ladderisation*, inspired by Montgomery Ladders. We present a semi-automatic tool-supported technique, aimed at the non-specialised developer, which refactors (a class of) C programs into functionally (and even algorithmically) equivalent counterparts with improved security properties. Our approach provides *refactorings* that transform the source code into its ladderised equivalent, driven by an underlying *verified rewrite system*, based on dependent types. Our rewrite system automatically finds rewritings of selected C expressions, facilitating the production of their equivalent ladderised counterparts for a subset of C. We demonstrate our approach on a number of representative examples from the cryptographic domain, showing increased security.

1 Introduction

Application security is often neglected by developers, who typically focus on functional aspects of their programs and timing optimisations. However, security attacks on software and devices are now commonplace as attackers exploit software vulnerabilities to access sensitive information. A type of

security attacks, which we address in this paper, is the *side channel attack (SCA)*, where attackers exploit irregularities of computations to expose secret keys. For example, if the branches of a conditional statement surrounding a predicate on a secret key are not regular [Koc96] (e.g. differ in execution time), attackers can use information leaked from the execution to determine which branche is executed and thus determine the secret key. SCAs are a particular problem as most developers lack the expertise to write their applications in a way that are secure against attackers. The most common techniques, if used at all, against SCAs typically involve *manually* measuring the program execution and then modifying the code in an *ad hoc* way, to try to reduce the amount of information leakage emitted by the execution, which might not even help against more sophisticated techniques like Simple [KJJ99] or Correlation [BCO04] Power Analysis. *Instead, developers need a structured, tool-supported way to ensure that their applications are more secure against SCAs.*

A Montgomery Ladder [JY03] is an algorithmic technique originally used for fast scalar multiplication on elliptic curves. It has been extended and generalised as a technique that applies to certain classes of algorithms, particularly those with a branching structure. A ladderised version of the algorithm is guaranteed to have more regular branching properties, and is therefore more secure against SCA. It is also more protected against other attacks like *fault injection* [ZAV04], because the interleaved variables prevent an attacker from gaining information on the secret key by comparing the output of the program between a normal and faulted execution.

In this paper, we introduce a new tool-supported technique that semi-automatically transforms portions of the code using refactoring and dependent types, resulting in a ladderised equivalent, with an increased security profile. Our technique is aimed at the non-specialised developer, where we provide high-level refactorings that transform the source code in a functionally (and even algorithmically [Mar19]) equivalent way, introducing a *semi-interleaved ladder* [MR20]. This refactoring process is aided by an underlying automated rewrite system, which uses dependent types to model a small arithmetic expression language. Dependent types allow us to show that our rewriting system is sound, giving confidence that the refactoring preserves the functionality of the code. Following application of the refactoring to a portion of code selected by the programmer, our rewriting system automatically searches for the correct rewriting to apply to the source code in order to introduce a semi-interleaved ladder via the refactoring tool. We demonstrate the effectiveness of our technique by showing that our refactoring approach can increase the security for a number of example applications with minimal development effort. Our contributions are:

1. we introduce a semi-automatic ladderisation technique for refactoring C programs into equivalent programs with improved security properties;
2. we introduce a novel algebraic rewriting system, using dependent types, for a small arithmetic expression language based on abelian rings, to find instances of semi-interleaved ladders;
3. we introduce a ladderisation refactoring for C that transforms `if-then` conditionals into semi-interleaved ladders, using our underlying rewrite system;
4. we demonstrate the effectiveness of our technique on two use-cases for security, showing that the ladderised variant is more protected against timing SCAs than the non-refactored variant.

2 Background

2.1 Montgomery Ladders

2.1.1 Modular Exponentiation

Let k be a secret key, and $k = \sum_{0 \leq i \leq d} k[i]\, 2^i$ its binary expansion of size $d+1$, i.e. $k[i]$ is the bit i of k. The *square-and-multiply* algorithm of Listing 1 computes the left-to-right modular exponentiation $a^k \bmod n$, using $a^{\sum_{0 \leq i \leq d} k[i] 2^i} = \prod_{0 \leq i \leq d} (a^{2^i})^{k[i]}$. This exponentiation is commonly used in cryptosystems like RSA [RSA78].

For every iteration, the multiplication ax is computed only if $k[i] = 1$, which can be detected by observing execution time [Koc96] or power profiles, e.g. by means of SPA (Simple Power Analysis) [KJJ99], and thus leads to information leakage from both time and power side-channel attacks. To

Listing 1: Square-and-multiply

```
1  // Most Significant Bit of an integer
2  #define MSB 8*sizeof(int) - 1
3
4  // i-th bit of integer k
5  int bit(int k, int i) {
6      return (k & (1 << i)) ? 1 : 0;
7  }
8
9  // square-and-multiply
10 int exp-sqmul(int a, int k, int n) {
11     int x = 1;
12     for (int i = MSB ; i >= 0 ; i--) {// left
       to right bits of k
13         x = x*x % n;
14         if (bit(k,i) == 1) {// current bit is a
       1
15             x = x*a % n;// leakage
16         }
17     }
18     return x;// = a^k % n
19 }
```

Listing 2: Square-and-multiply always

```
1  // square-and-multiply-always
2  int exp-sqmul-always(int a, int k, int n) {
3      int x = 1;
4      int y;
5      for (int i = MSB ; i >= 0 ; i--) {// left
       to right bits of k
6          x = x*x % n;
7          if (bit(k,i) == 1) {// current bit is 1
8              x = x*a % n;
9          } else {// current bit is a 0
10             y = x*a % n;// dummy operation
11         }
12     }
13     return x;// = a^k % n
14 }
```

Listing 3: Montgomery ladder (modular exponentiation)

```
1  // Montgomery ladder
2  int exp-ladder(int a, int k, int n) {
3      int x = 1;
4      int y = a % n;// y = a*x
5      for (int i = MSB ; i >= 0 ; i--) {// left
       to right bits of k
6          if (bit(k,i) == 1) {// current bit is 1
7              x = x*y % n;
8              y = y*y % n;
9          } else {// current bit is 0
10             y = y*x % n;
11             x = x*x % n;
12         }
13     }
14     return x;// = a^k % n
15 }
```

prevent SPA, regularity of the modular exponentiation algorithms is required, which means that both branches of the sensitive conditional branching perform the same operations, independently from the value of the exponent. Thus, an `else` branch is added with a dummy instruction [Cor99] in the *square-and-multiply-always* algorithm given in Listing 2, demonstrating a trade-off between execution time (or power usage) and security. However, countermeasures developed against a given attack may benefit another [SMKLM02]. Since the multiplication in the `else` branch of this algorithm is a dummy operation, a fault injected [ZAV04] in the register containing ax will eventually propagate through successive iterations and alter the final result only if $k[i] = 1$, thus leaking information. Therefore, an attacker is able to inject a fault in a given register at a given iteration, obtaining the digits of the secret key by comparing the final output with or without fault. This technique is known as a safe-error attack. This is not the case in the algorithm proposed by Montgomery [LM87] and given in Listing 3, where a fault injected in one register will eventually propagate to the other, and thus will alter the final result, preventing the attacker from obtaining information. The Montgomery ladder is algorithmically equivalent [Mar19] to the square-and-multiply(-always) algorithm(s), in the sense that x has the same value for every iteration. The invariant $y = ax$ is satisfied for every iteration, and can be used for self-secure exponentiation countermeasures [Gir06].

The `else` branch in Listing 3 is identical to the `then` branch, except that x and y are swapped, which also provides protection against timing and power leakage. Moreover, the variable dependency makes these variables interleaved, so this exponentiation is algorithmically (but only partially [MR20]) protected against safe-error attacks. As opposed to square-and-multiply-always in Listing 2, the code in the `else` branch is not dead, so will not be removed by compiler optimisations.

Listing 4: Double-and-add

```
1  // double-and-add
2  int scal-dbladd(int k, point a) {
3      point x = pt0;
4      for (int i = MSB ; i >= 0 ; i--) {// left
       to right bits of k
5          x = ptDbl(x);
6          if (bit(k,i) == 1) {// current bit is 1
7              x = ptAdd(a, x);// leakage
8          }
9      }
10     return x;// = k * a
11 }
```

Listing 5: Montgomery ladder (scalar multiplication)

```
1  // Montgomery ladder for the scalar
       multiplication
2  int scal-ladder(int k, point a) {
3      point x = pt0;
4      point y = a;
5      for (int i = MSB ; i >= 0 ; i--) {// left
       to right bits of k
6          if (bit(k,i) == 1) {// current bit is 1
7              x = ptAdd(x, y);// x + y
8              y = ptDbl(y);// 2y
9          } else {// current bit is 0
10             y = ptAdd(y, x);//y + x
11             x = ptDbl(x);// 2x
12         }
13     }
14     return x;// = k * a
15 }
```

2.1.2 Scalar Multiplication

Another prototypical example of a Montgomery ladder is the scalar multiplication used in Elliptic Curve Cryptography (ECC). ECC, independently introduced in 1985 by [Kob87] and [Mil86], is nowadays considered as an excellent choice for key exchange or digital signatures, especially when running on resource-constrained devices. The security of most cryptocurrencies is based on ECC, which has been standardized by the NIST [BCR+18, NIS13].

Let p be a prime. An elliptic curve in short Weierstrass form over a finite field $\mathbb{F}_p$ is defined by the set $E(\mathbb{F}_p) = \{(a, b) \in \mathbb{F}_p \times \mathbb{F}_p \mid b^2 = a^3 + \alpha a + \beta\} \cup \{o\}$ with $\alpha, \beta \in \mathbb{F}_p$ satisfying $4\alpha^3 + 27\beta^2 \neq 0$ and o being called the point at infinity. This set of points is an additive abelian group, where point addition is denoted $x + y$ and point o is the identity element. The point doubling is denoted $2x = x + x$.

The main operation in ECC is scalar multiplication $ka = a + \cdots + a$, where a is a point and k is an integer. It can be performed by using the *double-and-add* algorithm in Listing 4. The Montgomery ladder for the scalar multiplication provided in Listing 5 is similar to the Montgomery ladder for the modular exponentiation in Listing 3.

2.2 Dependent Types

Dependently-typed languages, such as Idris [Bra13], allow types to depend on any value. This enables properties to be expressed at the type level and proofs of properties as values of types, where both are verified by type-checking [Bra17a]. Dependently typed languages take advantage of the Curry-Howard correspondence, which states that, given a suitably rich type system, (certain kinds of) proofs can be represented as programs [SB17]. For languages with insufficiently rich type systems, such as C, dependently-typed languages can be used to produce an *abstract interpretation* [CC77] of a given program in those languages. Such abstract interpretations can be used to derive proofs of desired properties [AAPH12]. In the case of dependently-typed languages, under the propositions as types view, dependent types are used to represent predicates [Uni13]. For example, `(Even : (n : Nat) -> Type)` defines the type of evidence (or proofs) that a natural number, `n`, is even. In cases where the property does *not* hold true, e.g. `Even 1`, and assuming a suitably restricted definition of that property, the type is uninhabited. An uninhabited type represents falsity. Evidence that a predicate does not hold true can be represented by the type function, `(Not a = a -> Void)`, where `a` is a type variable and `Void` is the empty type (i.e. it has no constructors). Using dependent types in this way, properties that represent a (non-)functional specification can be encoded as predicates (i.e. types). Accordingly, total functions, `f : A -> B`, allow for the derivation of evidence that the predicate `B` can be constructed given evidence of `A`. Type-checking ensures the soundness of these functions [SKH+19]. The syntax of Idris is similar to Haskell [Hut07], and like Haskell, Idris supports algebraic data types with pattern matching, type classes, and do-notation. Unlike Haskell, Idris evaluates its terms eagerly.

Definitions, e.g. of languages and well-formedness, are provided as types in Idris. In this paper, we assume the reader is familiar with dependent types and Idris, for more details, see [Bra17b].

2.3 Abelian Rings

We will base the semantics of our arithmetic expression language on the theory of abelian rings. Algebraic structures describe operations over sets [LB99]. They can be seen as a generalisation of basic arithmetic operations, and thus facilitate techniques that are not intrinsically tied, for example, to a specific representation of integers. A *ring* $R_c = (c, \oplus, \otimes, \ominus, \mathbf{0}, \mathbf{1})$ is a carrier set, c, with addition, multiplication, negation, and additive and multiplicative identities. Addition forms an abelian group, i.e. where $\oplus$ is associative and commutative, $\mathbf{0}$ is an element in c such that $\forall x \in c,\ x \oplus \mathbf{0} = x$, and $\ominus$ is a unary inverse operator such that $\forall x \in c,\ x \oplus (\ominus x) = \mathbf{0}$. Similarly, multiplication forms a monoid, i.e. where $\otimes$ is associative and $\mathbf{1}$ is an element in c such that $\forall x \in c,\ x \otimes \mathbf{1} = x$. Finally, multiplication distributes over addition, i.e. $\forall x, y, z \in c,\ x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z) \wedge (x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$. *Abelian rings* add a requirement that multiplication is commutative.

2.4 Refactoring

Refactoring is the process of modifying the structure of a program while preserving its functional behaviour [OJ92]. The term was first introduced in [OJ92] in 1992, and the concept goes back at least to the fold/unfold system proposed by [BD77] in 1977. The main aims of refactoring are to increase code reuse, code quality, and programming productivity. At early stage, most refactoring was performed manually with the help of text editor "search and replace". In the last couple of decades, a range of refactoring tools have become available for various programming languages, that aid the programmer by offering a selection of automatic refactorings. Unlike the general program transformations, refactoring focusses on purely structural changes rather than on changes to program functionality, and is generally applied semi-automatically (i.e. under programmer direction), rather than fully automatically. This allows programmer knowledge to be exploited, and so permits a wider range of possible transformations.

Listing 6: Left-to-right exponentiation

```
1  // left-to-right exponentiation
2  int exp-sqmul-var(int a, int k, int n) {
3      int x = 1;
4      for (int i = MSB ; i >= 0 ; i--) {
5          if (bit(k,i) == 1) {
6              x = a*x*x % n;
7          } else {
8              x = x*x % n;
9          }
10     }
11     return x;
12 }
```

3 Ladderisation: Principles and Theory

This section builds upon *semi-interleaved* ladders, a generalisation of Montgomery Ladders (see Section 2.1). We introduce a ladderisation theorem where we take a non-ladderised program containing an iteration with an `if-then` conditional (i.e. without `else`), and show how it can be transformed into a semi-interleaved ladderised equivalent, via a system of equations. This theorem is used by the rewriting system of Section 4 and the refactoring framework of Section 5.

3.1 Ladder Equations

Marquer and Richmond [MR20] showed that the algorithm in Listing 1 can be rewritten as the common left-to-right exponentiation in Listing 6. More generally, they studied programs with *iterative conditional branching*, e.g. Listing 7. Whilst this approach does not depend on the number/depth of the considered loops, we focus on the case of one simple iteration in this paper. Assuming the conditional branching uses only a fresh variable x, a univariate iterative conditional branching with two unary functions θ (for the `then` branch) and ε (for the `else` branch) is obtained.

In order to prevent information leakage from side-channels or fault injections, another variable y is used in the algorithm described in Listing 8. An algorithm is defined to be *semi-ladderisable* if there

Listing 7: Iterative conditional branching

```
1  // Iterative conditional branching
2  int iter-cond-branch(int k, int init) {
3      int x = init;
4      for (int i = MSB ; i >= 0 ; i--) {
5          if (bit(k,i) == 1) {
6              x = theta(x);
7          } else {
8              x = epsilon(x);
9          }
10     }
11     return x;
12 }
```

Listing 8: Semi-Interleaved ladders

```
1  // Semi-Interleaved ladders
2  int SIL(int k, int init) {
3      int x = init;
4      int y = l(init);
5      for (int i = MSB ; i >= 0 ; i--) {
6          if (bit(k,i) == 1) {
7              x = f(x, y);
8              y = epsilon(y);
9          } else {
10             y = f(y, x);
11             x = epsilon(x);
12         }
13     }
14     return x;
15 }
```

exists a unary function ℓ and a binary function f such that, for every considered value x:

$$\begin{cases} \varepsilon(\ell(x)) = \ell(\theta(x)) & (1) \\ f(x, \ell(x)) = \theta(x) & (2) \\ f(\ell(x), x) = \ell(\varepsilon(x)) & (3) \end{cases}$$

Marquer and Richmond also proved, that if an algorithm is semi-ladderisable, then for every iteration of the ladderised algorithm the invariant $y = \ell(x)$ is satisfied. Moreover x has the same value for every iteration as in Listing 7, i.e. in the `then` branch, x is updated to $\theta(x)$ and in the `else` branch, x is updated to $\varepsilon(x)$, so the ladderised algorithm is not only functionally but algorithmically [Mar19] equivalent. Marquer and Richmond also introduced fully-interleaved ladders with an additional function and a more complex system of equations. Here, we focus on finding automatic solutions for the semi-interleaved ladders, which are more tractable.

Listing 9: if-then case

```
1  // special case of iterative conditional
       branching
2  int SIL_ifThen(int k, int init) {
3      int x = init;
4      for (int i = MSB ; i >= 0 ; i--) {
5          x = epsilon(x);
6          if (bit(k,i) == 1) {
7              x = lambda(x);
8          }
9      }
10     return x;
11 }
```

Thus, the ladderisation problem states that from given functions θ and ε, find functions ℓ and f satisfying (if possible) Equations (1) to (3). For instance, the left-to-right exponentiation algorithm (Listing 6) with initial functions $\theta(x) = ax^2$ and $\varepsilon(x) = x^2$ is ladderisable by using the functions $\ell(x) = ax$ and $f(x, y) = xy$, thus obtaining the Montgomery ladder (Listing 3).

3.2 The if-then Case

Solving the ladderisation problem in the general case poses a significant challenge; we therefore focus on a special case in this paper. Note that in the initial square-and-multiply algorithm (Listing 1) the `else` function $\varepsilon(x) = x^2$ (Line 13) and the invariant function $\ell(x) = ax$ (Line 15) are written explicitly. Because there is no `else` branch, the `else` function is called before the conditional branching, and the `then` function is actually $\theta(x) = \ell(\varepsilon(x))$.

Therefore, by restricting ourselves to the `if-then` case in Listing 9 we assume two initial functions ε and λ, and by replacing ℓ by λ and θ by $\lambda \circ \varepsilon$ in Equations (1) to (3) we obtain the following *if-then ladder equations*:

$$\begin{cases} \varepsilon(\lambda(x)) = \lambda(\lambda(\varepsilon(x))) & (4) \\ f(x, \lambda(x)) = \lambda(\varepsilon(x)) & (5) \\ f(\lambda(x), x) = \lambda(\varepsilon(x)) & (6) \end{cases}$$

Note that $\varepsilon(\lambda(x)) = \lambda(\lambda(\varepsilon(x)))$ does not depend on the unknown function f and depends only on initial functions ε and λ. Thus, whether Equation (4) is satisfied or not can be verified before searching for an appropriate function f. If not, the algorithm is not ladderisable. Otherwise, in Section 4.4 we

$$
\begin{array}{lcll}
e, e_i, e_1, e_2 \cdots \in \mathsf{AExp}_c & ::= & n & \text{Literals} \\
& \mid & x & \text{Variables} \\
& \mid & -e & \text{Additive Inverse} \\
& \mid & e^2 & \text{Square} \\
& \mid & e_1 + e_2 & \text{Addition} \\
& \mid & e_1 \times e_2 & \text{Multiplication}
\end{array}
$$

Figure 1: AExp_c syntax.

$$
\begin{array}{rcl}
\mathcal{A}[\![n]\!]s & = & n \\
\mathcal{A}[\![x]\!]s & = & s\ x \\
\mathcal{A}[\![-e]\!]s & = & \ominus \mathcal{A}[\![e]\!]s \\
\mathcal{A}[\![e^2]\!]s & = & \mathcal{A}[\![e]\!]s \otimes \mathcal{A}[\![e]\!]s \\
\mathcal{A}[\![e_1 + e_2]\!]s & = & \mathcal{A}[\![e_1]\!]s \oplus \mathcal{A}[\![e_2]\!]s \\
\mathcal{A}[\![e_1 \times e_2]\!]s & = & \mathcal{A}[\![e_1]\!]s \otimes \mathcal{A}[\![e_2]\!]s
\end{array}
$$

Figure 2: Semantic function $\mathcal{A}$ for AExp_c.

use $f(x, \lambda(x)) = \lambda(\varepsilon(x)) = f(\lambda(x), x)$ to try to construct a solution for f.

4 Algebraic Rewrites with Dependent Types

We define a rewriting system, over an arithmetic expression language that models only a small subset of C expressions, because we only need to reason about the functions identified (and potentially derived) as part of the ladderisation process. Our rewriting system forms an equivalence relation over expressions and is used to produce proofs that a given set of functions conform to the definition of semi-interleaved ladders (i.e. comprising conformity to Equations (4) to (6)). Our rewriting system enables the automatic derivation of f from a given ε and λ, is implemented in the dependently-typed language Idris, and is proof-carrying. All definitions correspond to data type definitions in our implementation. Proofs are represented via functions over those data types.

4.1 Syntax

We define the syntax of our arithmetic expression language, AExp_c, in Figure 1. AExp_c comprises literals, variables, negation, squaring, addition, and multiplication. The precise type of literals (e.g. integers or elliptic curves) is given by the carrier type, c, which must be equipped with an abelian ring, R_c. We assume a finite set of variables, $\mathcal{V}$, for each set of f, ε, and λ definitions. We use $a, x, y, z, \ldots$ to range over variables. For clarity, we will say that all sub-expressions x in $\lambda(x) = e$, refer to the argument to λ. Similarly, all sub-expressions x and y in $f(x, y) = e$ refer to their respective arguments to f.

Example 1. Given the modular exponentiation definition in Listing 1, where c is the set of integers $\mathbb{Z}$, we represent the corresponding f, ε, and λ in AExp_c below.

$$
\begin{array}{rcl}
\varepsilon(x) & = & x^2 \\
\lambda(x) & = & a \times x \\
f(x, y) & = & x \times y
\end{array}
$$

Here, a is a (free) variable, and $\mathcal{V} = \{a, x, y\}$.

Example 2. Given the point-scalar multiplication example in Listing 4, the corresponding f, ε, and λ in AExp_c are:

$$
\begin{array}{rcl}
\varepsilon(x) & = & x + x \\
\lambda(x) & = & x + a \\
f(x, y) & = & x + y
\end{array}
$$

Here, the carrier type, c, is defined to be the set of points of an elliptic curve in short Weierstrass form. As in Example 1, a is a (free) variable and constant, and $\mathcal{V} = \{a, x, y\}$.

In our implementation, AExp_c is represented by the type family, `AExp : (c : Type) -> (nvars : Nat) -> Type`. In order to simplify our representation, variables are encoded by finite sets, where `nvars` is the upper bound.

4.2 Semantics

In this section, we define a denotational semantics for AExp_c. AExp_c domain is defined as the abelian ring R_c, which equips the carrier type c. Figure 2 gives the semantic function for AExp_c, i.e. $\mathcal{A} : \mathsf{AExp}_c \to (\mathcal{V} \to c) \to R_c$. The state denoted $s : \mathcal{V} \to c$ is a total function from variables to literals. Addition, multiplication, and additive inverse are mapped to their respective operations in R_c. No

squaring operator exists in R_c, so e^2 is mapped to $\mathcal{A}[\![e]\!]s$ multiplication by itself.

Example 3. Given the modular exponentiation functions in Example 1, and given the state

$$s = \{a \mapsto 42, x \mapsto 5, y \mapsto 7\}$$

the semantic function $\mathcal{A}$ provides the following denotations:

$$\begin{array}{rcl} \mathcal{A}[\![x^2]\!]s & = & 25 \\ \mathcal{A}[\![a \times x]\!]s & = & 210 \\ \mathcal{A}[\![x \times y]\!]s & = & 35 \end{array}$$

We represent $\mathcal{A}$ by the family of types, `SmAExp : (e : AExp c nvs) -> (f : CRingExp {c} r nvs)`. Here, `CRingExp` is a family of types representing expressions in R_c, which can be reified given some concrete state, s; `r` corresponds to R_c. Each clause of $\mathcal{A}$ is encoded as a constructor to `SmAExp`.

4.3 Rewrites

We define the binary relation $\rightsquigarrow \subseteq \mathsf{AExp}_c \times \mathsf{AExp}_c$ to be a set of rewrites over AExp_c; $\overset{*}{\leftrightsquigarrow}$ to be the smallest reflexive transitive symmetric closure of $\rightsquigarrow$; and $\overset{*}{\leftrightsquigarrow}$ such that it alows rewriting subexpressions, e.g. in x^2, both x and x^2 itself can be rewritten. Since the exact set of rewrites necessary to prove that a set of equations determines a semi-interleaved ladder will vary with respect to the equations themselves, we parameterise our approach by $\rightsquigarrow$.

Example 4. Given the modular exponentiation functions in Example 1, the definition of $\rightsquigarrow$ in Figure 3 contains the rewrites that are sufficient in order to prove that modular exponentiation is an example of a semi-interleaved ladder.

We cannot assume that $\overset{*}{\leftrightsquigarrow}$ will be confluent or terminating; it is more likely that these two properties do not hold. For example, in the presence of a rewrite reflecting commutativity of addition and/or multiplication, $\overset{*}{\leftrightsquigarrow}$ is non-terminating.

Intuitively, $\rightsquigarrow$ is informed by the underlying algebraic structure (i.e. R_c). However, $\rightsquigarrow$ need not be limited to a bijection with the defining characteristics of that algebraic structure. We do however require that each rewrite rule preserves the semantics of the expression being rewritten, i.e. $\forall r \in \rightsquigarrow . \forall x, y \in \mathsf{AExp}_c . \forall s \in (\mathcal{A} \to R_c) . x\ r\ y \Leftrightarrow \mathcal{A}[\![x]\!]s = \mathcal{A}[\![y]\!]s$. This ensures that $\overset{*}{\leftrightsquigarrow}$ also preserves the semantics of the rewritten expression, and thus is a true equivalence relation with respect to the semantics of AExp_c.

In our implementation, we represent $\rightsquigarrow$ by the family of types, `ARewrite : (e, f : AExp c nvars) -> Type`, and similarly, $\overset{*}{\leftrightsquigarrow}$ by `Rewrite : (e,f : AExp c nvars) -> Type`. To simplify our representation, we do not index `Rewrite` by a generic type representing $\rightsquigarrow$, instead `ARewrite` is intended to be defined according to the desired definition of $\rightsquigarrow$. The definition of `Rewrite` includes constructors representing: rewrites from `ARewrite`, reflexivity, symmetry, and transitivity; in order to enable rewrites of subexpressions, it also includes three constructors (namely, `CongU` for unary operators, and both `CongBinL` and `CongBinR` for binary operators). For example, given some expression $-e$, and some rewriting of $e \overset{*}{\leftrightsquigarrow} f$, `CongU` is used to express the rewriting $-e \overset{*}{\leftrightsquigarrow} -f$. Soundness of rewrites is proven via the total function,

```
1  soundRewrite : (e,f  : AExp c nvs)
2               -> (x,y  : CRingExp r nvs)
3               -> (s1   : SmAExp e x) -> (s2 :
       SmAExp f y)
4               -> (r    : Rewrite e f) -> EquivRE x
        y
```

where `EquivRE` is a type family representing the definition $\forall s.\mathtt{x}s = \mathtt{y}s$, where $\mathtt{x} = \mathcal{A}[\![e]\!]$ and $\mathtt{y} = \mathcal{A}[\![f]\!]$.

4.4 Proof Search

In order to derive both f and the concomitant proofs for the semi-interleaved ladder equations (4) to (6) from given definitions of λ and ε, we use in Algorithm 1 a standard breadth-first search algorithm extended with a tabu-list. Since $\overset{*}{\leftrightsquigarrow}$ may not be confluent or terminating, our search procedure generates the tree as it searches for either a given expression (thus producing a proof that two expressions are equivalent) or a given subexpression (thus deriving f and proofs for the semi-interleaved ladder equations).

Algorithm 1 is customised by GenChildren() and Selection() operators. Given an expression, GenChildren() generates a set containing all expressions that result from applying all possible rewrites in $\overset{*}{\leftrightsquigarrow}$ (sans those derived via transitivity; expressions derived via transitivity, i.e. sequences of rewrites, are generated by descending the tree). The exact definition of the Selection() operator depends on the

$$\rightsquigarrow = \begin{cases} e^2 \rightsquigarrow e \times e & \text{Square expansion} \\ e_1 + e_2 \rightsquigarrow e_2 + e_1 & \text{Commutativity } (+) \\ e_1 \times e_2 \rightsquigarrow e_2 \times e_1 & \text{Commutativity } (\times) \\ e_1 + (e_2 + e_3) \rightsquigarrow (e_1 + e_2) + e_3 & \text{Associativity } (+) \\ e_1 \times (e_2 \times e_3) \rightsquigarrow (e_1 \times e_2) \times e_3 & \text{Associativity } (\times) \end{cases}$$

Figure 3: Definition of atomic rewrites $\rightsquigarrow$ for Example 4

Algorithm 1 Breadth-First Search with Tabu List

Require: $\tau = \emptyset$
Require: $d \geq 1$
Require: A queue, Q
Require: An expression, e_0
  Q.enqueue($(e_0, \texttt{id})$)
  $\tau = \tau \cup e_0$
  for $i = 0$ **to** d **do**
    $(e, r) \leftarrow Q$.dequeue()
    if Selection(e) **then**
      return (e, r)
    else
      $\tau = \tau \cup e$
      $C \leftarrow e$.GenChildren()
      for all $(e_i, r_i) \in C$ **do**
        if $e_i \notin \tau$ **then**
          Q.enqueue((e_i, r_i))
        end if
      end for
    end if
  end for
  return Nothing

application of Algorithm 1. To determine whether two expressions are equivalent, Selection() is the (propositional) equality operator. To derive f, the Selection() operator returns true when the generated expression, e, has $\lambda(x)$ as a subexpression, and when λ, ε, and f, which is derived from e, conform to the semi-interleaved ladder equations. In such cases, e represents $f(x, \lambda(x))$, from which we derive (by generalisation) f itself.

Since the generated tree may be infinitely large, we bound the search depth. When determining the equivalence of two expressions, e_1 and e_2, the result of Algorithm 1 is either Nothing (cannot find a proof that the two expressions are equivalent within the set depth) or the tuple comprising e_2 and the proof

Listing 10: discoF function

```
1  discoF : (depth : Nat) -> ErrorOr
2    (f ** (Rewrite (epsilon (lambda (Var varX)))
3              (lambda (lambda (epsilon (Var varX))))
         ,
4           Rewrite (lambda (epsilon (Var varX))) f,

5           SubTerm (lambda (Var varX)) f,
6           (f' ** Eq6 f f')))
7  discoF d = searchSt d (Var varX) (epsilon (
       lambda (Var varX))) (lambda (lambda (
       epsilon (Var varX)))) (lambda (epsilon (Var
        varX))) (lambda (Var varX))
```

of equivalence (i.e. the sequence of rewrites from e_1 to e_2). Similarly, when deriving f, Algorithm 1 either produces Nothing (cannot derive an f within the set depth) or the definition of f with proofs of Equations (4) to (6).

In our implementation, we do not produce a proof that Selection() does not pass for all nodes in the tree in order to optimise the search procedure, although this is possible in principle. We similarly remark that other search techniques could, in principle, be used in lieu of Algorithm 1.

4.5 Deriving f

This section gives an overview of the derivation of f and its constituent proofs of Equations (4) to (6) and a summary of the Idriss implementation of our deriviation[1].

The decision procedure, `discoF`, which generates proofs of Equations (4) to (6) is given in Listing 10, which takes a `depth` as a parameter to the breadth-first search. The search result is either an error (when no f can be derived) or a derived `f`, and proofs

[1] Available at `https://github.com/adbarwell/TeamPlayLadders`

Listing 11: Proof of Equation (6)

```
1  data Eq6 : (f : AExp c nvs) -> (f' : AExp c nvs)
2           -> Type where
3     MkEq6 : (p1:  ElemT lam f) -> (p2 : ElemT x f
          ')
4        -> toElemSimple p1 = toElemSimple p2
5        -> (p4 : ElemT x f) -> (p5 : ElemT lam f')
6        -> toElemSimple p4 = toElemSimple p5
7        -> Rewrite f f' -> Eq6 f f'
```

of the three equations. Proof of Equation (4) is given on Line 4-5, via the proof that $\varepsilon(\lambda(x)) \overset{*}{\rightsquigarrow} \lambda(\lambda(\varepsilon(x)))$. Proof of Equation (5) is given on Line 6-7, where $\exists f.\ \lambda(\varepsilon(x)) \overset{*}{\rightsquigarrow} f \wedge \lambda(x) \in f$ (i.e. we can rewrite $\lambda(\varepsilon(x))$ into f and $\lambda(x)$ is a subterm of f). Proof of Equation (6) is given on Line 8-9, where we find an f' such that the relation `Eq6 f f'` holds.

The type `Eq6` is given in Listing 11, where the type is formed by indexing two terms, `f` and `f'`. The idea behind the proof of Equation (6) is that the equation states that $f(\lambda(x), x) = \lambda(\varepsilon(x))$. For this we need to (simultaneously) substitute all occurrences of the term $\lambda(x)$ in our derived f for x and all occurrences of x for $\lambda(x)$. Given that we have already obtained a proof of Equation (5), we know that $f = \lambda(\varepsilon(x))$. As Equation (6) states that $f' = \lambda(\varepsilon(x))$, we swap the occurrences of $\lambda(x)$ and x within f to obtain some f'. The constructor `MkEq6` on Line 4 gives this relation, and its proof is formed via a decision procedure. First we give a proof of the positions of all the occurrences of the subterm $\lambda(x)$ in the derived f on Line 5 via the `ElemT` type (given in Listing 12). In a similar way, we give a proof of the position of all occurrences of x in f' on Line 6. Line 7 takes a proof that these positions are equivalent (i.e. that all occurrences of $\lambda(x)$ in f are the same as all occurrences of x in f'). We then take proofs of all occurrences of x in f (Line 8) and $\lambda(x)$ in f' (Line 9) and a proof of their equivalence on Line 10. Equivalence is via the `toElemSimple : ElemT t g -> ElemTermSimple` function, which simply projects an `ElemTerm t g` onto a version with the indexing removed, thus giving a structural equivalence over two terms. Given these proofs and a proof that $f \overset{*}{\rightsquigarrow} f'$, we can give a proof of Equation (6).

Listing 12 shows the type for `ElemT`, where it follows the standard convention for proving membership via a generalised form of `Elem`. The type is

Listing 12: Proof of the Occurrences of subterm `t` in `g`

```
1  data ElemT : (t : AExp c nvs) -> (g : AExp c
       nvs) -> Type where
2    HereT : ElemT t t
3    NotHereT : ElemT t y
4    BinOpT : (p : ElemT t e1) -> (q : ElemT t e2)
5          -> ElemT t (BinOp op e1 e2)
6    BinOpTR : (p : ElemT t e2) -> ElemT t (BinOp
       op e1 e2)
7    UniOpT : (there : ElemT t e1) -> ElemT t (
       UniOp op e1)
```

Listing 13: The decision procedure, searchSt

```
1  searchSt : (Show c, DecEq c) => {nvs : Nat} ->
       (d : Nat)
2          -> (a,b : AExp c nvs) -> (e,l : AExp c
       nvs)
3          -> ErrorOr (g ** (Rewrite a b, Rewrite
       e g,
4                            SubTerm l g, (h **
       Eq6 g h)))
```

indexed by terms `t` and `g` and gives a relation specifying the membership (and positions) of all occurrences of `t` in `g`. We omit details of the proof and decision procedure here to save space but they can be found in our implementation. Listing 13 shows the Idris function `searchSt`, giving the decision procedure for proofs of Equations (4) to (6) on Line 6-7. The function takes `d` to indicate the depth of the search; the terms `a` and `b` are used to derive proofs of Equation (4) (given on Line 6 via `Rewrite a b`); some `e`, and a proof of Equation (5) (via `Rewrite e g` on Line 6, along with a proof that `l` is a `SubTerm` of `g`); and the subterm `l`, for $\lambda(x)$. Proof of Equation (6) is given via a value of `Eq6`.

5 Ladderisation via Refactoring

This section describes several refactorings that enable an `if-then` iterative conditional branching program to be transformed into its ladderised equivalent. The refactoring process described here combines traditional program transformations such as *function extraction* and *function folding* (e.g. in the style of [BD77]), and a new refactoring technique

Listing 14: Abstract Interpretation in Idris of ε and λ

```
1  ||| epsilon(x) = x^2 mod n
2  epsilon : (x : AExp SInt 3) -> AExp SInt 3
3  epsilon x = UniOp Square x
4  ||| lambda(x) = a*x mod n
5  lambda : (x : AExp SInt 3) -> AExp SInt 3
6  lambda x = BinOp Mult (Var varA) x
```

that uses the algebraic rewriting from Section 4 to ladderise the program *semi-automatically*, introducing a semi-interleaved ladder. In this paper, to deal with security aspects, we extend an existing refactoring tool for C and C++ that was originally introducing parallel skeletons to sequential code bases [JBM+16]. The result of the refactoring process is an equivalent C program with improved security properties.

Shaping for Ladderisation The first step is a *function extraction* phase, which identifies from the source code the ε and λ functions that will be used to ladderise the code in the later steps. An example of this process is illustrated in Figure 4. Once ε and λ have been extracted, an *abstract interpretation* of the functions is given to the algebraic rewriting system from Section 4, resulting is a dependent pair, where the first element is the rewriting, and the second is a proof the rewriting is sound (see Listing 15).

Ladderisation Refactoring The ladderisation refactoring takes a C source file, with extracted ladder functions ε and λ, and transforms it into a ladderised version, after requiring the user to highlight a loop to ladderise. Figure 5 shows an example of the refactoring. The original program is shown on the left, in Figure 5a, where we assume the developer has highlighted the loop within the function `exp-sqmul` on Lines 14-19, which contains calls to the functions `epsilon` and `lambda` defined on Lines 3 and 7, and their corresponding call sites within the original program on Lines 15 and 17. The names of these functions are largely irrelevant: the refactoring analyses the structure of the highlighted function (`exp-sqmul`) to confirm it is a ladderisable software component. Figure 5b shows the ladderisation refactoring result. The `exp-sqmul` is transformed in a number of ways to introduce the semi-interleaved ladder. A call to the function `lambda` is made on Line 17, passing in `x`, `a`, and `n` as arguments; the result of this call is assigned to a new variable, `y`. It was observed that a standard `if-then-else` structure generates in assembly code a conditional jump performed only if the condition is false, thus costing more time in the `else` branch than in the `then` branch, leading to an unexpected timing imbalance in the ladderised variant. We thus instead use an `if-then; if not-then` structure in the refactoring. A call to `f` is made on Line 20 and `epsilon` on Line 21. Note that the semi-interleaved ladder also reverses the assignment of `x` and `y` in the second branch, where `y` is assigned to the result of the call to `f`, and `x` to `epsilon`. Listing 14 shows an example of a generated abstract interpretation in Idris of the functions `epsilon` and `lambda` from Figure 5b. The `epsilon` function, `x * x % n`, is represented on Line 8 as the `Square` of `x`, where the variable `x` has type `AExp SInt 3`, i.e. an expression of integer type, and 3 is the upper bound for the finite set of variables defined in scope. Note here that we do not capture the representation of the modulus operator from the original expression; this is due to the fact that we can omit the modulus as the commutative ring semantics from Section 4 handles the modulus implicitly. Similarly, the `lambda` function, `a * x % n`, is defined on Line 13 by the multiplication of `varA` by `x`. Listing 15 shows the proof of the rewriting to find f, comprising a dependent pair of the function f, and proofs of Equations (4) to (6).

6 Experimental Results

This section shows both our ladderisation refactoring and the security protection gained from it, comparing timing leakages between baseline and ladderised variants, for modular exponentiation in Section 6.1 and modular multiplication in Section 6.2. We gather cycle-accurate timing data from the ARM SystemC Cycle Model[2] for Cortex-M0. Unfortunately, modulus is computed by default with a function call to a library containing a costly and noisy implementation that leaks a lot of information . To avoid this, in Listing 16, we use a more secure

[2] `https://developer.arm.com/tools-and-software/simulation-models/cycle-models/arm-systemc-cycle-models`

```
1  // before function extraction
2  ...
3  // square-and-multiply
4  int exp-sqmul(int a, int k, int n) {
5      int x = 1;
6      for (int i = MSB ; i >= 0 ; i--) {
7          x = x*x % n;
8          if (bit(k,i) == 1) {
9              x = x*a % n;
10         }
11     }
12     return x;
13 }
14 ...
```

(a) Before

```
1  // after function extraction
2  ...
3  int epsilon(int x, int n) {
4     return x * x % n;
5   }
6
7  int lambda(int x, int a, int n) {
8      return x * a % n;
9   }
10
11 int exp-sqmul(int a, int k, int n) {
12     int x = 1;
13     for (int i = MSB ; i >= 0 ; i--) {
14         x = epsilon(x, n);
15         if (bit(k,i) == 1) {
16             x = lambda(x, a, n);
17         }
18     }
19     return x;
20 }
21 ...
```

(b) After

Figure 4: Extracting functions ε and λ into `epsilon` and `lambda`. Highlighted expressions in (a) are transformed in (b).

```
1  // before ladderisation
2  ...
3  int epsilon(int x, int n) {
4     return x * x % n;
5  }
6
7  int lambda(int x, int a, int n) {
8     return x * a % n;
9  }
10
11 ...
12 int exp-sqmul(int a, int k, int n) {
13     int x = 1;
14     for (int i = MSB ; i >= 0 ; i--) {
15         x = epsilon(x, n);
16         if (bit(k,i) == 1) {
17             x = lambda(x, a, n);
18         }
19     }
20     return x;
21 }
22 ...
23 ...
```

(a) Before

```
1  // after ladderisation
2  ...
3  int epsilon(int x, int n) {
4     return x * x % n;
5  }
6
7  int lambda(int x, int a, int n) {
8     return x * a % n;
9  }
10
11 int f(int x, int y, int n) {
12    return x * y % n;
13 }
14
15 int exp-sqmul(int a, int k, int n) {
16    int x=1, i;
17    int y = lambda(x, a, n);
18    for (i = MSB ; i >= 0 ; i--) {
19          if (bit(k,i) == 1) {
20                x = f(x, y, n);
21                y = epsilon(y, n);
22          }
23          if (bit(k,i) != 1) {
24                y = f(y, x, n);
25                x = epsilon(x, n);
26          }
27    }
28    return x;
29 }
30 ...
```

(b) After

Figure 5: Before and after applying the ladderisation refactoring. The highlighted loop in (a) is transformed into (b).

Listing 15: Proof of the rewriting to find f, which is a dependent pair with function f and proofs of Equations (4) to (6)

```
1  (MkDPair (BinOp Mult (Var FZ) (BinOp Mult (Var
       (FS (FS FZ))) (Var FZ)))
2    (Trans (Trans (Trans (Trans (Fun SqIsMult) (
       Sym (Fun (Associative MultIsAssoc)))) (
       CongB2 (Fun (Commutative MultIsComm)))) (
       CongB2 (Sym (Fun (Associative MultIsAssoc))
       ))) (CongB2
3        (CongB2 (Sym (Fun SqIsMult)))),
4      (Trans (Trans (Trans (Fun (Commutative
         MultIsComm)) (CongB1 (Fun SqIsMult))) (Sym
         (Fun (Associative MultIsAssoc)))) (CongB2 (
         Fun (Commutative MultIsComm))),
5        (BinOpRight Refl,
6      MkDPair (BinOp Mult (BinOp Mult (Var (FS (FS
         FZ))) (Var FZ)) (Var FZ)) (MkEq6 (BinOpT
         NotHereT HereT) (BinOpT NotHereT HereT)
         Refl (BinOpT HereT
7        NotHereT) (BinOpT HereT NotHereT) Refl (Fun
           (Commutative MultIsComm)))))))
```

and efficient modulus operation implemented as a Barrett's reduction [MvOV96], where *shift* is precomputed as the smallest integer s such that $n < 2^s$, and mu as $\frac{2^{2\times \text{shift}}}{n}$ rounded down. Being precomputed and only read in the code, these parameters have no impact on refactoring and analysis.

6.1 Modular Exponentiation

The square-and-multiply program in Section 2.1.1 is adapted in Listing 17 for the timing analysis. The analysis tools uses the `loopbound` pragma so that the iteration is done 32 times. For every iteration i, `k & (1 << i)` computes bit i of the secret key k. Listing 18 is the refactoring result detailed in Section 5, where we applied a manual *unfolding* [BD77] (i.e. inlining) for this paper. In both programs the number of modular squarings is constant, but in Listing 17 the number of modular multiplications depends on $\text{HW}(k)$ the Hamming weight (i.e., the number of 1s) of the key, while in Listing 18 it is constant. Thus the execution time is expected to be linear in $\text{HW}(k)$ for Listing 17 but constant for Listing 18. In the crypto-system RSA [RSA78], the modulus is a product $n = pq$ of distinct prime numbers that we generated where every key is prime with Euler's totient $\phi(n) = (p-1)(q-1)$. For a given n we precomputed the corresponding *shift* and

Listing 16: Barrett's reduction

```
1  int mod_barrett(unsigned int v, unsigned int n,
       unsigned int shift, unsigned int mu) {
2      unsigned int dummy = v, r;
3      r = v - (((v >> (shift - 1)) * mu) >> (
       shift + 1))*n;
4      // require at most 2 more subtractions
5      if (r < n)
6          dummy = dummy - n;// dummy operation
7      if (r >= n)
8          r = r - n;
9      if (r < n)
10         dummy = dummy - n;// dummy operation
11     if (r >= n)
12         r = r - n;
13     return r;// = v % n
14 }
```

Listing 17: Square-and-multiply

```
1  unsigned int x = 1;
2  _Pragma("loopbound min 32 max 32");
3  for (i = 31 ; i >= 0 ; i--) {// left to right
       bits of k
4      x = mod_barrett(x*x, n, shift, mu);
5      if ((k & (1 << i)) != 0) {
6          x = mod_barrett(a*x, n, shift, mu);
7      }
8  }
```

mu values for the Barrett's reduction [MvOV96]. Finally, we randomly generated values for a such that $1 < a < n$. Our timing observations (in clock cycles), obtained from the ARM simulator, correspond to the expected complexity:

$$\text{time}_{\text{sqmul}}(k) = 35 \times \text{HW}(k) + 1357$$
$$\text{time}_{\text{ladder}} = 2899$$

We experimentally confirmed that execution time does not depend on public values a and n, and that different keys with the same Hamming weight produce the same execution time, thus the time analysis depends only on the Hamming weight and not the actual value of the key. We used 33 keys with Hamming weights from 0 to 32, 4 values for a, and 4 values for n for the plots in Figure 6. Therefore, an attacker can infer the Hamming weight of the secret key from the observation of execution time for the square-and-multiply (unprotected) variant, but can obtain no information from the ladderised (protected) variant because it is constant-time.

Listing 18: Ladderised (and inlined) exponentiation

```
1  unsigned int x = 1;
2  unsigned int y = a;
3  _Pragma("loopbound min 32 max 32");
4  for (i = 31 ; i >= 0 ; i--) {// left to right
       bits of k
5      if ((k & (1 << i)) != 0) {
6          x = mod_barrett(x*y, n, shift, mu);
7          y = mod_barrett(y*y, n, shift, mu);
8      }
9      if ((k & (1 << i)) == 0) {
10         y = mod_barrett(y*x, n, shift, mu);
11         x = mod_barrett(x*x, n, shift, mu);
12     }
13 }
```

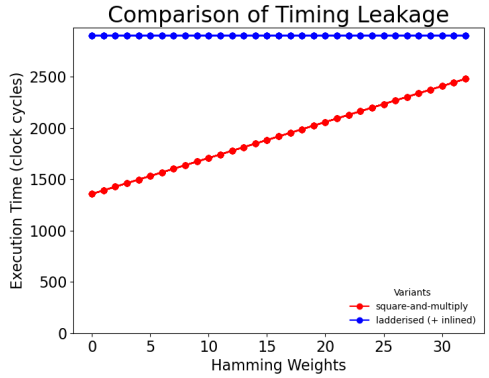

Figure 6: Modular Exponentiation

6.2 Modular Multiplication

Implementing data structures to efficiently deal with elliptic curves is not this paper focus, so we demonstrate the ladderisation refactoring on the scalar multiplication example by using modular integers, thus a modular multiplication. The double-and-add program from Section 2.1.2 has been adapted in Listing 19 for the timing analysis. Listing 20 is the inlined product of the ladderisation refactoring. Since integer addition and multiplication cost the same on Cortex-M0, we obtain the same formula for execution time as in Section 6.1. We use the same inputs to plot Figure 7, demonstrating again a timing leakage for the double-and-add (unprotected) variant, and a constant-time ladderised (protected) variant.

Listing 19: Square-and-multiply

```
1  unsigned int x = 0;
2  _Pragma("loopbound min 32 max 32");
3  for (i = 31 ; i >= 0 ; i--) {// left to right
       bits of k
4      x = mod_barrett(2*x, n, shift, mu);
5      if ((k & (1 << i)) != 0) {
6          x = mod_barrett(a+x, n, shift, mu);
7      }
8  }
```

Listing 20: Ladderised (and inlined) multiplication

```
1  unsigned int x = 0;
2  unsigned int y = a;
3  _Pragma("loopbound min 32 max 32");
4  for (i = 31 ; i >= 0 ; i--) {// left to right
       bits of k
5      if ((k & (1 << i)) != 0) {
6          x = mod_barrett(x+y, n, shift, mu);
7          y = mod_barrett(2*y, n, shift, mu);
8      }
9      if ((k & (1 << i)) == 0) {
10         y = mod_barrett(y+x, n, shift, mu);
11         x = mod_barrett(2*x, n, shift, mu);
12     }
13 }
```

7 Related Work

Maruyama [Mar07] proposes *secure refactorings*, aimed at transforming Java programs, by introducing a number of small transformations that increase security by exploiting Object Oriented properties, such as introducing immutability, etc. It does not address the problem of SCAs. Mumtaz et al. [MAMN18] propose a technique using refactoring to eliminate bad code smells that can contribute to security vulnerabilities. Rather than introducing refactorings specific to security, the authors propose a combination of existing refactorings to address the issue. Abid et al. [AKA+20] propose a similar technique, studying the impact of different refactorings on a number of static security metrics. However, these papers do not consider security issues such as side-channel attacks, or ladderisation as a technique to eliminate the vulnerability. To defend against SCAs, algorithms based on the square-and-multiply-always have been proposed [BNP07] by checking invariants [KKRS16] violated if a fault is injected. Joye [Joy07, Joy09] introduces highly regular right-

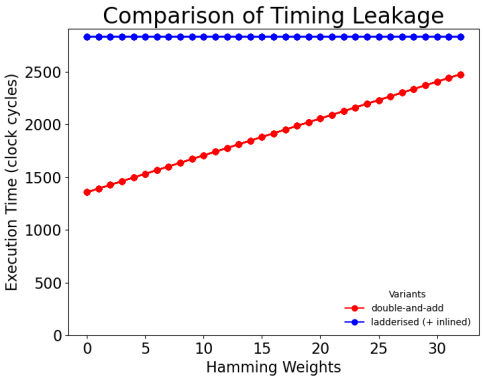

Figure 7: Modular Multiplication

to-left variants and left-to-right/right-to-left variants respectively, and Walter [Wal17] demonstrates their duality. Marquer and Richmond [MR20] abstract away the algorithmic strength of the Montgomery ladder against SCAs, by defining semi- and fully-ladderisable programs. Brown et al. [BBM+19] propose a Contract Specification Language (CSL), to allow the developer to capture non-functional properties (NFPs) about their program (including time and energy). CSL also provides the developers with mechanisms to write assertions over these NFPs, with an underlying formal abstract model using dependent types to provide proofs of correctness for the assertions in the form of contracts. A proof of decidability is produced if the contract has been met. Barwell and Brown [BB19] extended CSL to provide implementations of the underlying proof-engine, modelling the specification of the contracts for a representative subset of C. They both define an improved abstract interpretation, together with a sound operational semantics that automatically derives proofs of assertions, and define inference algorithms for the derivation of both abstract interpretations and the context over which the interpretation is indexed. However, both these approaches only target time (and energy) as the primary NFP, rather than security properties and SCAs.

8 Conclusions and Future Work

This paper present a new tool-supported technique to semi-automatically refactor C programs into functionally (and even algorithmically) equivalent versions with fewer security vulnerabilities. We use an underlying algebraic rewriting system, using dependent types, for a small arithmetic expression language based on abelian rings, finding instances of semi-interleaved ladders. We introduced a ladderisation refactoring for C that transforms `if-then` conditionals into semi-interleaved ladders, using our underlying rewrite system. This balancing of conditional branches decreases or removes non-functional leakages, preventing an attacker from obtaining information on the sensitive data. We presented our technique on two cryptographic examples: modular exponentiation and multiplication. In both cases our technique refactors program parts so that they are executed in constant time, removing timing vulnerability leakage, demonstrating that our refactoring approach increases the security of a number of applications with minimal developer effort.

Limitations: The rewriting and refactoring system presented here handles a set of examples that are common in the RSA community, based on a semantics of abelian rings. Whilst we present a small set of syntactic constructs and rewrite rules, limiting the practical application, our approach can be extended to handle a wider range of examples. Although any extension will require the modification of most definitions in our implementation, this comprises the addition of new cases to those definitions; the general framework remains the same. Given a sufficiently large extension, it is possible, in principle, to represent C expressions in full. The primary concern in extending the system is with regard to the efficiency of the (proof) search algorithm. Both a larger language and set of rewrite rules will result in larger search trees. We conjecture that this could be mitigated via the use of alternative search algorithms, whilst keeping the selection procedure the same, to ensure that our proof search remains both tractable and scalable.

Future work: We will extend our system to process the generalised case of conditional expressions in C (e.g. the `if-then-else` case, rather than just the `if-then` case). We will also extend the system

to deal with nested loops, fully interleaved ladders, and multivariate cases. Further extensions to our work would be in modelling a more extensive subset of C semantics, including additional C constructs, and even generalising the technique to other languages and paradigms, e.g. Java, Python, or Haskell. We will explore the soundness properties further by giving general soundness proofs of the refactorings themselves and also further validate our technique on a full and tractable set of benchmarks and use-cases. Finally, we will explore using refactoring techniques to reduce the security risk of programs in general. This may include more refactorings to prevent SCAs, or applying refactoring to other security risks such as fault-injection, or high-level structural risks caused by anti-patterns.

ACKNOWLEDGEMENTS

The authors thank Nicolas Kiss (Inria Rennes) for the timing measurements. This work was generously supported by EU Horizon 2020 project *TeamPlay* (`https://www.teamplay-h2020.eu`), grant #779882, and UK EPSRC *Energise*, EP/V006290/1.

References

[AAPH12] Elvira Albert, Puri Arenas, Germán Puebla, and Manuel V. Hermenegildo. Certificate size reduction in abstraction-carrying code. *TPLP*, 12(3):283–318, 2012.

[AKA+20] Chaima Abid, Marouane Kessentini, Vahid Alizadeh, Mouna Dhouadi, and Rick Kazman. How does refactoring impact security when improving quality? a security-aware refactoring approach. *IEEE Transactions on Software Engineering*, 2020.

[BB19] Adam Barwell and Christopher Brown. A Trustworthy Framework for Resource-Aware Embedded Programming. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19)*. ACM, 2019.

[BBM+19] Christopher Brown, Adam D. Barwell, Yoann Marquer, Céline Minh, and Olivier Zendra. Type-driven verification of non-functional properties. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, PPDP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.

[BCR+18] Elaine B. Barker, LiLy Chen, Allen L. Roginsky, Apostol Vassilev, and Richard Davis. *Recommendation for Pair-Wise Key Establishment Using Dicrete Logarithm Cryptography.* Number 800-56A Rev. 3 in Special Publication. NIST SP, April 2018.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[BNP07] Arnaud Boscher, Robert Naciri, and Emmanuel Prouff. CRT RSA Algorithm Protected Against Fault Attacks. In Damien Sauveron, Konstantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater, editors, *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, pages 229–243, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Bra13] Edwin Brady. Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[Bra17a] Edwin Brady. Type-driven development of concurrent communicating systems. *Computer Science*, 18(3):219–240, 2017.

[Bra17b] Edwin Brady. *Type-Driven Development with Idris*. Manning Publications Co., 2017.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[Cor99] Jean-Sébastien Coron. Resistance Against Differential Power Analysis For Elliptic Curve Cryptosystems. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems*, pages 292–302, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Gir06] C. Giraud. An RSA Implementation Resistant to Fault Attacks and to Simple Power Analysis. *IEEE Transactions on Computers*, 55(9):1116–1120, Sep. 2006.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, New York, NY, USA, 2007.

[JBM+16] Vladimir Janjic, Christopher Brown, K. Mackenzie, Kevin Hammond, Marco Danelutto, Marco Aldinucci, and José Daniel García. RPL: A domain-specific language for designing and implementing parallel C++ applications. In *PDP*, pages 288–295. IEEE Computer Society, 2016.

[Joy07] Marc Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 135–147, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Joy09] Marc Joye. Highly Regular m-Ary Powering Ladders. In Michael J. Jacobson, Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, pages 350–363, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[JY03] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 291–302, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[KKRS16] Ágnes Kiss, Juliane Krämer, Pablo Rauzy, and Jean-Pierre Seifert. Algorithmic Countermeasures Against Fault Attacks and Power Analysis for RSA-CRT. In François-Xavier Standaert and Elisabeth Oswald, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 111–129, Cham, 2016. Springer International Publishing.

[Kob87] N. Koblitz. Elliptic Curve Cryptosystems. *Math. Comp*, 48:243–264, 01 1987.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[LB99] S.M. Lane and G. Birkhoff. *Algebra*. AMS Chelsea Publishing Series. Chelsea Publishing Company, 1999.

[LM87] Peter L. Montgomery. Montgomery, P.L.: Speeding the Pollard and Elliptic Curve Methods of Factorization. Math. Comp. 48, 243-264. *Mathematics of Computation - Math. Comput.*, 48:243–243, 01 1987.

[MAMN18] Haris Mumtaz, Mohammad Alshayeb, Sajjad Mahmood, and Mahmood Niazi. An Empirical Study to Improve Software Security through the Application of Code Refactoring. *Information and Software Technology*, 96:112–125, 2018.

[Mar07] Katsuhisa Maruyama. Secure refactoring - improving the security level of existing code. In Joaquim Filipe, Boris Shishkov, and Markus Helfert, editors, *ICSOFT (SE)*, pages 222–229. INSTICC Press, 2007.

[Mar19] Yoann Marquer. Algorithmic Completeness of Imperative Programming Languages. *Fundamenta Informaticae*, 168(1):51–77, July 2019.

[Mil86] Victor S. Miller. Use of Elliptic Curves in Cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[MR20] Y. Marquer and T. Richmond. A hole in the ladder : Interleaved variables in iterative conditional branching. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 56–63, 2020.

[MvOV96] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[NIS13] Digital signature standard (DSS), July 2013. FIPS PUB 186-4, U.S.Department of Commerce/National Institute of Standards and Technology.

[OJ92] William Opdyke and Ralph Johnson. Refactoring object-oriented frameworks. 1992.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[SB17] Franck Slama and Edwin Brady. Automatically proving equivalence by type-safe reflection. In *CICM*, volume 10383 of *Lecture Notes in Computer Science*, pages 40–55. Springer, 2017.

[SKH+19] Christopher Schwaab, Ekaterina Komendantskaya, Alasdair Hill, Frantisek Farka, Ronald P. A. Petrick, Joe B. Wells, and Kevin Hammond. Proof-carrying plans. In *PADL*, volume 11372 of *Lecture Notes in Computer Science*, pages 204–220. Springer, 2019.

[SMKLM02] Yen Sung-Ming, Seungjoo Kim, Seongan Lim, and Sangjae Moon. A Countermeasure against One Physical Cryptanalysis May Benefit Another Attack. In Kwangjo Kim, editor, *Information Security and Cryptology — ICISC 2001*, pages 414–427, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[Wal17] C. D. Walter. The Montgomery and Joye Powering Ladders are Dual. *IACR ePrint Archive*, 1081:1–6, 2017.

[ZAV04] H. Ziade, R. Ayoubi, and R. Velazco. A Survey on Fault Injection Techniques. *International Arab Journal of Information Technology*, Vol. 1, No. 2, July:171–186, 2004.