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

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

Adam D. Barwell
a.barwell@imperial.ac.uk
University of St Andrews and
Imperial College London
UK

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

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

Tania Richmond
tania.richmond.nc@gmail.com
Inria, then DGA - Maîtrise de
l'Information
Rennes, France

Chen Gu
guchen@hfut.edu.cn
Hefei University of Technology
Hefei, China

ABSTRACT

Cyber attacks become more and more prevalent every day. An arms race is thus engaged between cyber attacks and cyber defences. 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 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. Using our tool-supported technique, we demonstrate our approach on a number of representative examples from the cryptographic domain, showing increased security.

CCS CONCEPTS

• **Software and its engineering** → *Semantics*; *Integrated and visual development environments*; • **Security and privacy** → *Intrusion/anomaly detection and malware mitigation.*

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

KEYWORDS

Security, Dependent Types, Idris, Soundness, Refactoring, Rewriting, Semantics, Side-channel attacks, Fault injection

ACM Reference Format:
Christopher Brown, Adam D. Barwell, Yoann Marquer, Olivier Zendra, Tania Richmond, and Chen Gu. October 2020. Semi-Automatic Ladderisation: Improving Code Security through Rewriting and Dependent Types. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Writing applications with security in mind is often a neglected activity amongst developers, who have typically focussed on functional aspects of their programs, and on timing optimisations. However, security attacks on software and devices are becoming more commonplace as attackers exploit vulnerabilities in the software to access sensitive information, such as secret keys or passwords. An example of such a security attack, which we address in this paper, is a *side channel attack*, 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 [24] (e.g. are dissimilar in execution time), attackers can use information leakage from the execution to determine which branches are executed in the program and consequently determine secret keys or sensitive data. Side-channel attacks are a particular problem as most developers lack the expertise to write their applications in a way that are secure against attackers. Indeed, the most common techniques to secure applications, if used at all, typically involve manually measuring the program execution and then modifying the code in an ad hoc way, in order to try to reduce the amount of information leakage emitted by the execution, which might not even help against more sophisticated techniques like Simple [23] or Correlation [10] Power Attacks. *Instead, what developers need is a structured, tool-supported way to ensure that their applications are more secure against side-channel attacks.*

A Montgomery Ladder [20] is an algorithmic technique originally used for fast scalar multiplication on elliptic curves. It has since 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 side-channel attacks. It is also more protected against other attacks like *fault injection* [40], 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 [27]) equivalent way, introducing a *semi-interleaved ladder* [28]. 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. The main contributions of this paper 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 side-channel attacks than the non-refactored variant.

2 BACKGROUND

In this section, we give an overview of the frameworks and techniques that we use in the paper. Section 2.1 introduces the concept of a *ladder*; Section 2.2 gives a short overview of dependent types and Idris; Section 2.3 gives a definition abelian rings; and, Section 2.4 gives an overview on refactoring.

2.1 Montgomery Ladders

In this section we introduce the concept of a ladder by using the prototypical example of Montgomery Ladders for modular exponentiation and scalar multiplication.

2.1.1 Modular Exponentiation. Let k be a secret key, and $k = \sum_{0 \leq i \leq d} k[i]\, 2^i$ be its binary expansion of size $d+1$, i.e. $k[i]$ is the bit i of k. The *square-and-multiply* algorithm given in Listing 1

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 }
```

computes the (left-to-right) modular exponentiation $a^k \bmod n$, by 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 crypto-systems like RSA [34].

For every iteration, the multiplication ax is computed only if $k[i] = 1$, which can be detected by observing execution time [24] or power profiles, e.g. by means of SPA (Simple Power Analysis) [23], and thus leads to information leakage from both time and power side-channel attacks. To 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 [13] in the *square-and-multiply-always* algorithm given in Listing 2, demonstrating a trade-off between execution time (or energy consumption) and security. However, countermeasures developed against a given attack may benefit another [37]. Since the multiplication in the `else` branch of this algorithm is a dummy operation, a fault injected [40] 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 [25] and given in Listing 3, where a fault injected in one register will eventually propagate

Listing 3: Montgomery ladder (modular exponentiation)

```
// Montgomery ladder
int exp-ladder(int a, int k, int n) {
    int x = 1;
    int y = a % n;// y = a*x
    for (int i = MSB ; i >= 0 ; i--) {// left to right bits of k
        if (bit(k,i) == 1) {// current bit is 1
            x = x*y % n;
            y = y*y % n;
        } else {// current bit is 0
            y = y*x % n;
            x = x*x % n;
        }
    }
    return x;// = a^k % n
}
```

Listing 4: Double-and-add

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

Listing 5: Montgomery ladder (scalar multiplication)

```
// Montgomery ladder for the scalar multiplication
int scal-ladder(int k, point a) {
    point x = pt0;
    point y = a;
    for (int i = MSB ; i >= 0 ; i--) {// left to right bits of k
        if (bit(k,i) == 1) {// current bit is 1
            x = ptAdd(x, y);
            y = ptDbl(y);
        } else {// current bit is 0
            y = ptAdd(y, x);
            x = ptDbl(x);
        }
    }
    return x;// = k * a
}
```

to the other, and thus will alter the final result, preventing the attacker from obtaining information. The Montgomery ladder is algorithmically equivalent [27] 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 [15].

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 [28]) protected against safe-error attacks. Finally, 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.

2.1.2 Scalar Multiplication. Another prototypical example of a Montgomery ladder is the scalar multiplication used in Elliptic Curve Cryptography (ECC). ECC was independently introduced in 1985 by Neal Koblitz [22] and Victor Miller [31]. It is nowadays considered as an excellent choice for key exchange or digital signatures, especially when these mechanisms run on resource-constrained devices. The security of most cryptocurrencies is based on ECC, which has been standardized by the NIST [1, 4].

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 [7], 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 [8]. 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 [36]. For languages with insufficiently rich type systems, such as C, dependently-typed languages can be used to produce an *abstract interpretation* [14] of a given program in those languages. Such abstract interpretations can be used to derive proofs of desired properties [3]. In the case of dependently-typed languages, under the propositions as types view, dependent types are used to represent predicates [38]. 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 [35]. The syntax of Idris is similar to Haskell [16], 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 defined by giving their definitions as types in Idris. In this paper, we assume the reader is familiar with dependent types and Idris. For those unfamiliar, full texts are available elsewhere [9].

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 [26]. 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 the additional requirement that multiplication is commutative.

2.4 Refactoring

Refactoring is the process of modifying the structure of a program while preserving its functional behaviour [33]. The term refactoring was first introduced by Opydyke [33] in 1992, and the concept goes back at least to the fold/unfold system proposed by Burstall and Darlington [12] in 1977. The main aims of refactoring are to increase code quality, programming productivity, and code reuse, which leads to increased productivity and programmability. Historically, most refactoring was performed manually with the help of text editor "search and replace" facilities at early stage. However, in the last couple of decades, a diverse 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, e.g. about safety properties, to be exploited, and so permits a wider range of possible transformations.

3 LADDERISATION: PRINCIPLES AND THEORY

In this section, we build upon *semi-interleaved* ladders, which are a generalisation of Montgomery Ladders (defined in Section 2). We introduce a ladderisation theorem where we can take a non-ladderised program containing an iteration with a `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. The theorem presented here is used by the algebraic rewriting system of Section 4 and the refactoring framework presented in Section 5.

3.1 Ladder Equations

Marquer and Richmond [28] showed that the algorithm in Listing 1 can be rewritten as the common left-to-right exponentiation in Listing 6. More generally, Marquer and Richmond studied programs with *iterative conditional branching*, e.g. Listing 7. Whilst this approach does not depend on the number/depth of the considered

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 }
```

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 }
```

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 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

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 }
```

$\theta(x)$ and in the `else` branch, x is updated to $\varepsilon(x)$, so the ladderised algorithm is not only functionally but algorithmically [27] equivalent. Marquer and Richmond also introduced fully-interleaved ladders with an additional function and a more complex system of equations. However, in this paper, we focus on finding automatic solutions for the semi-interleaved ladders, which are more tractable.

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 Subsection 4.4 we use $f(x, \lambda(x)) = \lambda(\varepsilon(x)) = f(\lambda(x), x)$ to try to construct a solution for f.

4 ALGEBRAIC REWRITES USING DEPENDENT TYPES

In this section, we define a rewriting system over an arithmetic expression language. Since it is only necessary to reason about the functions identified (and potentially derived) as part of the ladderisation process, our arithmetic expression language models only a small subset of the expressions found in C. 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)). Additionally, our rewriting system enables the automatic derivation of f from a given ε and λ. Our rewriting system is implemented in the dependently-typed language, Idris, and is proof-carrying. Accordingly, all definitions in this section correspond to data type definitions in our implementation. Similarly, proofs are represented via functions over those data types.

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

Figure 1: AExp_c syntax.

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 4.1. 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 4.2. Given the point-scalar multiplication example in Listing 4, we represent the corresponding f, ε, and λ in AExp_c below.

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

Here, the carrier type, c, is defined to be the set of points of an elliptic curve in short Weierstrass form. As in Example 4.1, 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.

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

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

4.2 Semantics

In this section, we define a denotational semantics for AExp_c. The domain of AExp_c is defined to be 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 the additive inverse are mapped to their respective operations in R_c. Since there is no squaring operator in R_c, e^2 is mapped to the multiplication of $\mathcal{A}[\![e]\!]s$ by itself.

Example 4.3. Given the modular exponentiation functions in Example 4.1, and given the state

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

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

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

In our implementation, $\mathcal{A}$ is represented by the family of types,

```
SmAExp : (e : AExp c nvars)
       -> (f : CRingExp {c} ring nvars)
       -> Type
```

Here, `CRingExp` is a family of types representing expressions in R_c, which can be reified given some concrete state, s; `ring` 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. Similarly, we define $\overset{*}{\leftrightsquigarrow}$ to be the smallest reflexive transitive symmetric closure of $\rightsquigarrow$. Additionally, we define $\overset{*}{\leftrightsquigarrow}$ such that it permits the rewriting of 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 4.4. Given the modular exponentiation functions in Example 4.1, the below definition of $\rightsquigarrow$ contains the rewrites that are sufficient in order to prove that modular exponentiation is an example of a semi-interleaved ladder.

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

Note that we cannot assume that $\overset{*}{\leftrightsquigarrow}$ will be confluent or terminating; conversely, 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 semantic meaning of the expression that is 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 semantic meaning 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,

```
soundRewrite : (e,f  : AExp c nvars)
             -> (x,y  : CRingExp ring nvars)
             -> (s1   : SmAExp e x)
             -> (s2   : SmAExp f y)
             -> (r    : Rewrite e f)
             -> EquivRE x y
```

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

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 the breadth-first search technique in Algorithm 1. Algorithm 1 is 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 application of Algorithm 1. When the intention is to determine whether two expressions are equivalent, Selection() is the (propositional) equality operator. When the aim is to derive f, the Selection() operator returns true when the generated expression,

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, 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

Listing 10: discoF function

```
1  discoF : (depth : Nat)
2  -> ErrorOr
3    (f **
4      (Rewrite (epsilon (lambda (Var varX)))
5        (lambda (lambda (epsilon (Var varX)))),
6      Rewrite (lambda (epsilon (Var varX))) f,
7      SubTerm (lambda (Var varX)) f,
8      (f' **
9        Eq6 f f')))
10 discoF d =
11   searchSt d (Var varX) (epsilon (lambda (Var varX))) (lambda (
       lambda (epsilon (Var varX)))) (lambda (epsilon (Var varX)))
        (lambda (Var varX))
```

Listing 11: Proof of Equation (6)

```
1  data Eq6 : (f  : AExp c nvars)
2          -> (f' : AExp c nvars)
3          -> Type where
4    MkEq6 :
5          (p1:  ElemTerm lam f)
6       -> (p2 : ElemTerm x f')
7       -> toElemSimple p1 = toElemSimple p2
8       -> (p4 : ElemTerm x f)
9       -> (p5 : ElemTerm lam f')
10      -> toElemSimple p4 = toElemSimple p5
11      -> Rewrite f f'
12      -> Eq6 f f'
```

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 can derive (by generalisation) and return f itself.

Since the generated tree may be infinitely large, we bound the depth to which we search. When determining the equivalence of two expressions, e_1 and e_2, the result of Algorithm 1 is either Nothing (when we are unable find a proof that the two expressions are equivalent within the given depth) or the tuple comprising e_2 and the proof of equivalence (i.e. the sequence of rewrites that produces e_2 from e_1). Similarly, when deriving f, Algorithm 1 either produces Nothing (when we cannot derive an f within the given bounded 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

In this section, we give an overview of the derivation of f and its constituent proofs of Equations (4) to (6). The implementation of our deriviation is given in Idris, which we summarise here[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 result of the search is either an error (e.g. in the case that an f cannot be derived) or a derived `f`, and proofs 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 `ElemTerm` 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 : ElemTerm 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 `ElemTerm`, where it follows the standard convention for proving membership via a generalised form of `Elem`. The type is indexed by terms `t` and `g` and gives a relation specifying the membership (and positions) of all occurrences of `t` in `g`. We omit full details of the proof and decision procedure here due to space limitations but these can be found in our implementation.

[1]The source code is available at https://github.com/adbarwell/TeamPlayLadders

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

```
1  data ElemTerm : (t : AExp c nvars)
2                -> (g : AExp c nvars)
3                -> Type where
4    JustHereT : ElemTerm t t
5    JustNotHereT : ElemTerm t y
6    JustBinOpT : (there1 : ElemTerm t e1)
7               -> (there2 : ElemTerm t e2)
8               -> ElemTerm t (BinOp op e1 e2)
9    JustBinOpTRight : (there1 : ElemTerm t e2)
10              -> ElemTerm t (BinOp op e1 e2)
11   JustUniOpT : (there : ElemTerm t e1)
12              -> ElemTerm t (UniOp op e1)
```

Listing 13: The decision procedure, searchSt

```
1  searchSt : (Show c, DecEq c)
2  => {nvars : Nat}
3  -> (depth : Nat)
4  -> (a,b : AExp c nvars)
5  -> (e,l : AExp c nvars)
6  -> ErrorOr (g ** (Rewrite a b, Rewrite e g, SubTerm l g, (g' **
       Eq6 g g')))
```

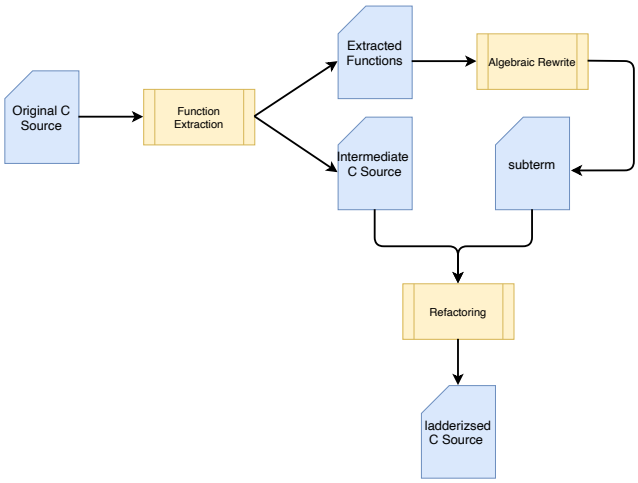

Figure 3: The workflow of refactoring framework for program ladderisation

Listing 13 illustrates the Idris function `searchSt`, giving the decision procedure for proofs of Equations (4) to (6) on Line 6-7. The function takes `depth` 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, alongwith 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 INTRODUCING LADDERISATION VIA REFACTORING

In this section we describe a number of refactorings that enable a `if-then` iterative conditional branching program to be transformed into its ladderised equivalent. The refactoring process described here employs a combination of traditional program transformations, such as *introduce new definition*, and *function folding* (e.g. in the style of Burstall and Darlington [12]), and also a new refactoring technique that uses the algebraic rewriting technique from Section 4 to ladderise the program *semi-automatically*, introducing a semi-interleaved ladder. In this paper we extend an existing refactoring tool for C and C++ that was originally targeted at introducing parallel skeletons to sequential code bases [17]; we extend the tool here to deal with security aspects. The result of the refactoring process is an equivalent C program with improved security properties.

5.1 Shaping for Ladderisation

An overview of the general refactoring workflow for ladderisation is illustrated in Figure 3. The programmer starts with their original C source file that they wish to ladderise. The first step is to go through a *function extraction* phase, which identifies the ε and λ functions from the source code that will be used to ladderise the code in the later steps. Once these functions have been identified and extracted, an *abstract interpretation* of the functions is given to the algebraic rewriting system that is defined in Section 4. The result of this rewriting is a dependent pair, where the first element is the rewriting, and the second is a proof the rewriting is sound. An example of such an output is given in Listing 15. The next step is for the tool to then refactor the original source code with the extracted functions into a ladderised version. An example of this ladderised version is shown in Figure 5b. Starting with the original C program, the developer first identifies the two functions, ε and λ, from the program and extracts them into new functions. Figure 4 illustrates an example of this step. The original program is shown on the left, with the refactored version on the right. The changes between the before and after version are highlighted in the figure in yellow. This extraction is achieved by using the built-in refactoring *Extract Function*, which takes a highlighted expression, and creates a new function, with its body being the expression. Any free variables of the expression are captured as function arguments. The original highlighted expression is then *folded* against the newly introduced function definition, passing the free variables of the extracted function's body as arguments. This step is a standard refactoring technique, but we modify it so that an abstract interpretation of the extracted functions is passed to the algebraic rewriting system from Section 4. An example of this process is illustrated in Figure 4, where the functions corresponding to ε and λ are selected by the developer and extracted into functions, by highlighting the expression `x * x % n` (Line 7) and `x * a % n` (Line 9) in the source code of Figure 4a and then using the *Extract Method* refactoring in Eclipse. The code after function extraction is shown in Figure 4b.

5.2 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 Line 14-19, with 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. Please note that the names of these functions are largely irrelevant: the refactoring proceeds by analysing the

```
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) != 0) {
20                y = f(y, x, n);
21                x = epsilon(x, n);
22          }
23          if (bit(k,i) == 0) {
24                x = f(x, y, n);
25                y = epsilon(y, 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).

structure of the highlighted function (`squaringExp`) to confirm it is a ladderisable software component. The result of the ladderisation refactoring is shown in Figure 5b. Here, the code has been transformed in a number of places. First, a new function, `f`, has been introduced on Line 11 corresponding to the output of the algebraic

Listing 14: Abstract Interpretation in Idris of ε and λ

```
1   export
2   varA : Fin 3
3   varA = FS (FS FZ)
4
5   ||| epsilon(x) = x^2 mod n
6   export
7   epsilon : (x : AExp SInt 3) -> AExp SInt 3
8   epsilon x = UniOp Square x
9
10  ||| lambda(x) = a*x mod n
11  export
12  lambda : (x : AExp SInt 3) -> AExp SInt 3
13  lambda x = BinOp Mult (Var varA) x
```

rewriting system (again, the choice of the name of `f` is chosen by the refactoring tool as a *fresh* function name). The `squaringExp` 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 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. In the *true* branch, `f` is called on Line 24, but the arguments are reversed, and then a call to `epsilon` with `y` rather than `x`. Note that the semi-interleaved ladder also reverses the assignment of `x` and `y` in the *true* branch: here `x` is assigned to the result of the call to `f`, and `y` 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. In Idris, the semantics of $*$ includes the modulus operation. In C, we need to distinguish it from the standard $*$ operator. Similary, 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

In this section, we demonstrate both our ladderisation refactoring and the security protection gained from it, showing baseline and ladderised variants, and comparing timing leakages between the two. We investigate modular exponentiation in Section 6.1 and modular multiplication in Section 6.2. We gather timing information by using a cycle-accurate timing data obtained from the ARM SystemC Cycle Model[2] for Cortex-M0. Unfortunately, the modulus is not a native operation, thus is computed by default by using a function call to a library containing a costly implementation that leaks a lot of information, and could thus interfere with our measurements. In Listing 16, we therefore use a more secure and efficient modulus operation implemented as a Barrett's reduction [30], 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. Since they are precomputed and only read in the studied code, these new parameters have no impact on the refactoring and the analysis.

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

Listing 15: Proof of the rewriting to find f, which is a dependent pair comprising the function f, with 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 (JustBinOpT JustNotHereT JustHereT) (
        JustBinOpT JustNotHereT JustHereT) Refl (JustBinOpT
        JustHereT
7       JustNotHereT) (JustBinOpT JustHereT JustNotHereT) Refl (Fun
        (Commutative MultIsComm)))))))
```

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  }
```

6.1 Modular Exponentiation

The square-and-multiply program presented in Section 2.1.1 is adapted in Listing 17 for the timing analysis. The `loopbound` pragma is used to indicate the analysis tools that the iteration is done 32 times. For every iteration i, `k & (1 << i)` computes bit i of the secret key k.

The program in Listing 18 is the result of the refactoring detailed in Section 5, where we have applied a manual *unfolding* [12] (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[3] of the key, while in Listing 18 it is

[3]The number of non-zero symbols in the representation. Because we use binary this is the number of 1s.

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  }
```

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 }
```

constant. Thus the execution time is expected to be linear in $\mathrm{HW}(k)$ for Listing 17 but constant for Listing 18. In the crypto-system RSA [34], the modulus is a product $n = pq$ of distinct prime numbers, that we generated such that every key is prime with Euler's totient number $\phi(n) = (p-1)(q-1)$. For a given n we precomputed the corresponding *shift* and *mu* values for the Barrett's reduction [30]. 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:

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

We confirmed experimentally that the execution time does not depend on the 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.

6.2 Modular Multiplication

Properly implementing the data structures required to deal efficiently with elliptic curves is not the focus of this paper, we therefore 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. The program in Listing 20 is the inlined product of the ladderisation refactoring. Because integer addition and multiplication cost the same for Cortex-M0, we obtained the same formula for the execution times

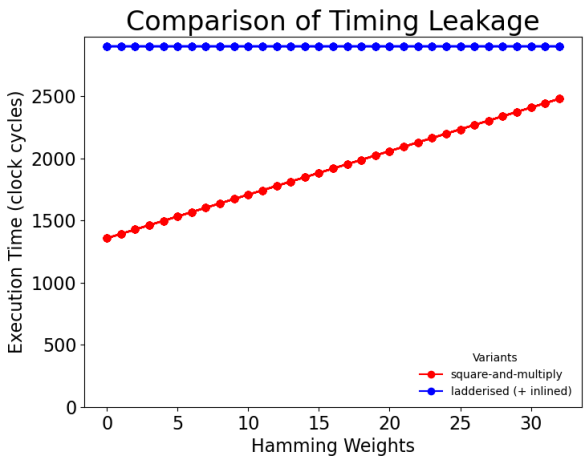

Figure 6: Modular Exponentiation

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 }
```

as in Section 6.1. We used 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.

7 RELATED WORK

Maruyama [29] proposes *secure refactorings*, aimed at transforming object-oriented Java programs. The paper aims to introduce 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 side-channel attacks. Mumtaz et al. [32] propose a technique to use refactoring to eliminate bad code smells that can contribute to security vulnerabilities. Rather than introducing refactorings specific to security, the authors propose using a combination of existing refactorings to address the issue.

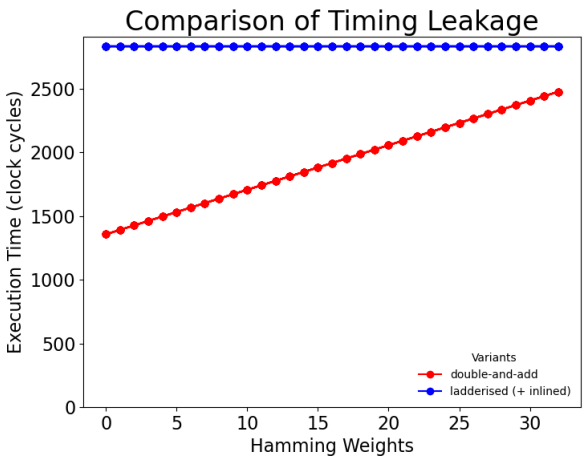

Figure 7: Modular Multiplication

Abid et al. [2] propose a similar technique, where they study the impact of different refactorings on a number of static security metrics. However, both of these do not consider security issues such as side-channel attacks, or use ladderisation as a technique to eliminate the vulnerability. In order to defend against side-channel attacks, algorithms based on the square-and-multiply-always have been proposed [6] by checking invariants [21] violated if a fault is injected. Joye [18, 19] introduces highly regular right-to-left variants and left-to-right/right-to-left variants respectively, and Walter [39] demonstrates their duality. Marquer and Richmond [28] abstract away the algorithmic strength of the Montgomery ladder against side-channel attacks, by defining semi- and fully-ladderisable programs. Brown et al. [11] propose a Contract Specification Language (CSL), to allow the developer to capture non-functional properties about their program (including time and energy). CSL also provides the developers with mechanisms to write assertions over these non-functional properties, 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 [5] extended CSL to provide implementations of the underlying proof-engine, modelling the specification of the contracts for a representative subset of C. Barwell and Brown define both 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 of these approaches only target time (and energy) as the primary non-functional property, rather than security properties and side-channel attacks.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new tool-supported technique to semi-automatically refactor C programs into functionally (and even algorithmically) equivalent versions with fewer security vulnerabilities. We used an underlying algebraic rewriting system, using dependent types, for a small arithmetic expression language based on abelian rings, to find instances of semi-interleaved ladders. Finally, we introduced a ladderisation refactoring for C that transforms `if-then` conditionals into semi-interleaved ladders, using our underlying rewrite system. This balancing of the 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 the programs so that they are executed in constant time, removing any timing vulnerability leakage, demonstrating that our refactoring approach increases the security of a number of applications with the minimal of developer effort. In the future, we plan to take our work forward in a number of new ways. We will extend our system to work over 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 the work presented here would be in modelling a more extensive set of the semantics of C, including additional C constructs, and even generalising the technique to other languages and paradigms, such as Java, Python, or Haskell. We will explore the soundness properties further by giving general soundness proofs of the refactorings themselves. We will also further validate our technique on a full and tractable set of benchmarks and use-cases. Finally, we will explore further the idea of using refactoring techniques to reduce the security risk of programs in general. This may include more refactorings to prevent side channel attacks, or applying refactoring to other kinds of security risks such as fault-injection, or even high-level, more structural risks caused by anti-patterns.

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 in order 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 can 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.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] 2013. Digital Signature Standard (DSS). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf FIPS PUB 186-4, U.S.Department of Commerce/National Institute of Standards and Technology.

[2] Chaima Abid, Marouane Kessentini, Vahid Alizadeh, Mouna Dhouadi, and Rick Kazman. 2020. How Does Refactoring Impact Security When Improving Quality? A Security-Aware Refactoring Approach. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/tse.2020.3005995

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

[4] Elaine B. Barker, LiLy Chen, Allen L. Roginsky, Apostol Vassilev, and Richard Davis. 2018. *Recommendation for Pair-Wise Key Establishment Using Dicrete Logarithm Cryptography.* Number 800-56A Rev. 3 in Special Publication. NIST SP. https://doi.org/10.6028/NIST.SP.800-56Ar3

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

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

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

[8] Edwin Brady. 2017. Type-driven Development of Concurrent Communicating Systems. *Computer Science* 18, 3 (2017), 219–240. https://doi.org/10.7494/csci.2017.18.3.1413

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

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

[11] Christopher Brown, Adam D. Barwell, Yoann Marquer, Céline Minh, and Olivier Zendra. 2019. Type-Driven Verification of Non-Functional Properties. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming* (Porto, Portugal) *(PPDP '19)*. Association for Computing Machinery, New York, NY, USA, Article 6, 15 pages. https://doi.org/10.1145/3354166.3354171

[12] R. M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *J. ACM* 24, 1 (1977), 44–67. https://doi.org/10.1145/321992.321996

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

[14] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

[15] C. Giraud. 2006. An RSA Implementation Resistant to Fault Attacks and to Simple Power Analysis. *IEEE Trans. Comput.* 55, 9 (Sep. 2006), 1116–1120. https://doi.org/10.1109/TC.2006.135

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

[17] Vladimir Janjic, Christopher Brown, K. Mackenzie, Kevin Hammond, Marco Danelutto, Marco Aldinucci, and José Daniel García. 2016. RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications. In *PDP*. IEEE Computer Society, 288–295.

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

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

[20] Marc Joye and Sung-Ming Yen. 2003. The Montgomery Powering Ladder. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, Burton S. Kaliski, çetin K. Koç, and Christof Paar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 291–302.

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

[22] N. Koblitz. 1987. Elliptic Curve Cryptosystems. *Math. Comp* 48 (01 1987), 243–264. https://doi.org/10.1090/S0025-5718-1987-0866109-5

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

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

[25] Peter L. Montgomery. 1987. Montgomery, P.L.: Speeding the Pollard and Elliptic Curve Methods of Factorization. Math. Comp. 48, 243-264. *Mathematics of Computation - Math. Comput.* 48 (01 1987), 243–243. https://doi.org/10.1090/S0025-5718-1987-0866113-7

[26] S.M. Lane and G. Birkhoff. 1999. *Algebra.* Chelsea Publishing Company.

[27] Yoann Marquer. 2019. Algorithmic Completeness of Imperative Programming Languages. *Fundamenta Informaticae* 168, 1 (July 2019), 51–77. https://doi.org/10.3233/FI-2019-1824

[28] Y. Marquer and T. Richmond. 2020. A Hole in the Ladder : Interleaved Variables in Iterative Conditional Branching. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*. 56–63. https://doi.org/10.1109/ARITH48897.2020.00017

[29] Katsuhisa Maruyama. 2007. Secure Refactoring - Improving the Security Level of Existing Code.. In *ICSOFT (SE)* (2009-02-24), Joaquim Filipe, Boris Shishkov, and Markus Helfert (Eds.). INSTICC Press, 222–229. http://dblp.uni-trier.de/db/conf/icsoft/icsoft2007-2.html#Maruyama07

[30] A. Menezes, P. van Oorschot, and S. Vanstone. 1996.. *Handbook of Applied Cryptography.* CRC Press. 816 pages.

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

[32] Haris Mumtaz, Mohammad Alshayeb, Sajjad Mahmood, and Mahmood Niazi. 2018. An Empirical Study to Improve Software Security through the Application of Code Refactoring. *Information and Software Technology* 96 (2018), 112–125. https://doi.org/10.1016/j.infsof.2017.11.010

[33] William Opdyke and Ralph Johnson. 1992. Refactoring Object-Oriented Frameworks. (1992).

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

[35] Christopher Schwaab, Ekaterina Komendantskaya, Alasdair Hill, Frantisek Farka, Ronald P. A. Petrick, Joe B. Wells, and Kevin Hammond. 2019. Proof-Carrying Plans. In *PADL (Lecture Notes in Computer Science, Vol. 11372)*. Springer, 204–220.

[36] Franck Slama and Edwin Brady. 2017. Automatically Proving Equivalence by Type-Safe Reflection. In *CICM (Lecture Notes in Computer Science, Vol. 10383)*. Springer, 40–55.

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

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

[39] C. D. Walter. 2017. The Montgomery and Joye Powering Ladders are Dual. *IACR ePrint Archive* 1081 (2017), 1–6. https://eprint.iacr.org/2017/1081.pdf

[40] H. Ziade, R. Ayoubi, and R. Velazco. 2004. A Survey on Fault Injection Techniques. *International Arab Journal of Information Technology* Vol. 1, No. 2, July (2004), 171–186. https://hal.archives-ouvertes.fr/hal-00105562