Automatic Ladderisation:
Improving Code Security through Rewriting and Dependent Types

Chris Brown
Adam Barwell, Yoann Marquer, Olivier Zandra, Tania Richmond, Chen Gu
Teamplay Workshop
May 2021
Security
Side-Channel Attacks

```c
// Most Significant Bit of an integer
#define MSB 8*sizeof(int) - 1

// i-th bit of integer k
int bit(int k, int i) {
    return (k & (1 << i)) ? 1 : 0;
}

// square-and-multiply
int exp-sm(int a, int k, int n) {
    int x = 1;
    for (int i = MSB ; i >= 0 ; i--) {
        x = x*x % n;
        if (bit(k,i) == 1) {
            x = x*a % n;
        }
    }
    return x;
}
```
```c
// Most Significant Bit of an integer
#define MSB 8*sizeof(int) - 1

// i-th bit of integer k
int bit(int k, int i) {
    return (k & (1 << i)) ? 1 : 0;
}

// square-and-multiply
int exp-sm(int a, int k, int n) {
    int x = 1;
    for (int i = MSB ; i >= 0 ; i--) {
        x = x*x % n;
        if (bit(k,i) == 1) {
            x = x*a % n;
        }
    }
    return x;
}
```
Side-Channel Attacks

```c
// square-and-multiply
int exp-sm(int a, int k, int n) {
    int x = 1;
    for (int i = MSB; i >= 0; i--) {
        x = x*x % n;
        if (bit(k,i) == 1) {
            x = x*a % n;
        }
    }
    return x;
}
```
Thinking “Secure”

• Fundamentally, programmers must learn to “think secure”
  • this requires new high-level programming constructs

• Programmers tune for “performance” rather than for security
  • Performance tuning can lead to security vulnerabilities!

• Common techniques involve manually measuring timing, then modify code in an ad-hoc way!

• You cannot secure code effectively without timing/energy information
  • this needs to be included as part of the design!
Secure Software is Difficult!
Tool-supported Security Framework

Original C Source → Function Extraction → Extracted Functions → Algebraic Rewrite

Extracted Functions → Intermediate C Source → subterm

Intermediate C Source → Refactoring

Refactoring → ladderized C Source
Refactoring

- Refactoring changes the structure of a program’s source code
  ...... using well-defined rules
  ... semi-automatically under programmer guidance
Security Refactoring

• Minimal effort
• Automated guidance
• Programmer in the loop
• Increases programmer productivity
• Expert and non-specialist programmers
• Portability and maintainability
• Resilience and robustness of code
• timing and power estimations
Security C++ Refactoring

- Integrated into Eclipse (CDT)
- Supports full C++(11) standard
- Layout and comment preserving
- Undoable
- Preview feature
Algebraic Rewriting

\[
\begin{align*}
\epsilon(\lambda(x)) &= \lambda(\epsilon(x)) \\
f(x, \lambda(x)) &= \lambda(\epsilon(x)) \\
f(\lambda(x), x) &= \lambda(\epsilon(x))
\end{align*}
\]

// Most Significant Bit of an integer
#define MSB 8*sizeof(int) - 1

// i-th bit of integer k
int bit(int k, int i) {
    return (k & (1 << i)) ? 1 : 0;
}

// square-and-multiply
int exp-sm(int a, int k, int n) {
    int x = 1;
    for (int i = MSB ; i >= 0 ; i--) {
        x = x*x % n;
        if (bit(k,i) == 1) {
            x = x*a % n;
        }
    }
    return x;
}

// no else branch in iterative conditional branching
int noelse(int k, int init) {
    int x = init;
    for (int i = MSB ; i >= 0 ; i--) {
        x = epsilon(x);
        if (bit(k,i) == 1) {
            x = lambda(x);
        }
    }
    return x;
}
Semi-Interleaved Ladder

```c
// Semi-Interleaved ladders
int SIL(int k, int init) {
    int x = init;
    int y = l(init);
    for (int i = MSB; i >= 0; i--) {
        if (bit(k,i) == 1) {
            x = f(x, y);
            y = epsilon(y);
        }
    else
    {
        y = f(y, x);
        x = epsilon(x);
    }
    return x;
}
```
Idris

- Functional Programming Language
- First Class Dependent Types
  - Functions can compute types
  - Types can contain values
- Developed at St Andrews
- A proof assistant using types
A Subset of C

de, e_i, e_1, e_2 \cdots \in \text{AExp}_c \\
::= \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\
\ | \\n
\text{n} \quad \text{Literals} \\
\text{x} \quad \text{Variables} \\
\text{-e} \quad \text{Additive Inverse} \\
\text{e^2} \quad \text{Square} \\
\text{e_1 + e_2} \quad \text{Addition} \\
\text{e_1 \times e_2} \quad \text{Multiplication}
Semantics

- Denotational semantics based on Abelian rings
  - Addition, multiplication, associativity, commutativity, transitive...

- Soundness properties ensure rewritings conform to semantic preservation
  - Dependent types allow proofs of soundness
Dependent Rewritings

\[ \text{ARewrite : } (e, f : \text{AExp \, c \, nvars}) \rightarrow \text{Type} \]

\[ \text{Rewrite : } (e, f : \text{AExp \, c \, nvars}) \rightarrow \text{Type.} \]

\[ \text{soundRewrite : } (e, f : \text{AExp \, c \, nvars}) \]
\[ \rightarrow (x, y : \text{CRingExp \, ring \, nvars}) \]
\[ \rightarrow (s_1 : \text{SmAExp \, e \, x}) \]
\[ \rightarrow (s_2 : \text{SmAExp \, f \, y}) \]
\[ \rightarrow (r : \text{Rewrite \, e \, f}) \]
\[ \rightarrow \text{EquivRE \, x \, y} \]

where \( \text{EquivRE} \) is a type family representing the definition \( \forall s. xs = ys \), where \( x = A[e] \) and \( y = A[f] \).
Finding a rewriting...

\[\begin{align*}
\rightsquigarrow &= \begin{cases}
  e^2 &\rightsquigarrow e \times e \\
n + e_2 &\rightsquigarrow e_2 + e_1 \\
n \times e_2 &\rightsquigarrow e_2 \times e_1 \\
n + (e_2 + e_3) &\rightsquigarrow (e_1 + e_2) + e_3 \\
n \times (e_2 \times e_3) &\rightsquigarrow (e_1 \times e_2) \times e_3
\end{cases}
\]

Square is self-multiplication
Commutativity (+)
Commutativity (×)
Associativity (+)
Associativity (×)

\[\forall r \in \rightsquigarrow \ . \ \forall x, y \in \text{AExp}_c \ . \ \forall s \in (A \rightarrow R_c) \ . \ x \rightsquigarrow y \iff A[x]s = A[y]s\]
Algorithm 1 Breadth-First Search with Tabu List

Require: \( \tau = \emptyset \)
Require: \( d \geq 1 \)
Require: A queue, \( Q \)
Require: An expression, \( e_0 \)

\[ Q\text{.enqueue}((e_0, \text{id})) \]
\[ \tau = \tau \cup e_0 \]

for \( i = 0 \) to \( d \) do

\( (e, r) \leftarrow Q\text{.dequeue}() \)
if Selection\((e)\) then

return \((e, r)\)
else

\( C \leftarrow e\text{.GenChildren}() \)
for all \((e_i, r_i) \in C\) do

if \( e_i \not\in \tau \) then

\( Q\text{.enqueue}((e_i, r_i)) \)
end if
end for
end if
end for

return Nothing
// Most Significant Bit of an integer
#define MSB 8*sizeof(int) - 1

// i-th bit of integer k
int bit(int k, int i) {
    return (k & (1 << i)) ? 1 : 0;
}

// square-and-multiply
int exp-sm(int a, int k, int n) {
    int x = 1;
    for (int i = MSB ; i >= 0 ; i--) {
        x = x*x % n;
        if (bit(k,i) == 1) {
            x = x*a % n;
        }
    }
    return x;
}
Function Extraction

(a) Before

```c
// before function extraction
...
int squaringExp(int a, int k, int n) {
    int x=1, i;
    for (i = MSB ; i >= 0 ; i--) {
        x = x * x % n;
        if (bit(k,i) == 1) {
            x = x * a % n;
        }
    }
    return x;
}
...
```

(b) After

```c
// after function extraction
...
int epsilon(int x, int n) {
    return x * x % n;
}

int lambda(int x, int a, int n) {
    return x * a % n;
}

int squaringExp(int a, int k, int n) {
    int x=1, i;
    for (i = MSB ; i >= 0 ; i--) {
        x = epsilon(x, n);
        if (bit(k,i) == 1) {
            x = lambda(x, a, n);
        }
    }
    return x;
}
...
```
Abstract Interpretation

```c
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
```

Generate an abstract representation of the C program to be used by the algebraic rewriting system.
1  Just ((x * y)
2       **
3   (CongB2(
4       CongB2( (Sym (SqIsMult))) )).
5   CongB2( (Sym (Asso(*))) ).
6   CongB2( Comm(*) ).
7   (Sym (Asso(*))).
8   SqIsMult,
9   (CongB2( Comm(*) ).
10   (Sym (Asso(*))).
11   CongB1( SqIsMult ).
12   Comm(*),
13   CongB2( (Sym (SqIsMult)) ).
14   (Sym (Asso(*))))
\[
f(x, y) = x \cdot y
\]
Rewriting and Proof

\[(a \times x)^2 = a \times (a \times (x^2))\]

\[(a \times x) \times (a \times x) = a \times (a \times (x^2))\] \(\text{SqIsMult}\)

\[a \times (x \times (a \times x)) = a \times (a \times (x^2))\] \(\text{sym Associative}\)

\[a \times (a \times (x \times x)) = a \times (a \times (x^2))\] \(\text{cong Associative}\)

\[a \times (a \times (x^2)) = a \times (a \times (x^2))\] \(\text{cong (cong (sym SqIsMult))}\)

\[\varepsilon(x) = x^2\]

\[\lambda(x) = a \times x\]

\[f(x, y) = x \times y\]
Rewriting and Proof

\[ x^* (a^*) = a^* x^2 \]
\[ (a^*)^* x = a^* x^2 \text{ } \{ \text{Commutative} \} \]
\[ a^* (x^* x) = a^* x^2 \text{ } \{ \text{sym Associative} \} \]
\[ a^* x^2 = a^* x^2 \text{ } \{ \text{cong (sym SqIsMult)} \} \]
Rewriting and Proof

(a*x)*x = a*x^2

\( a*(x*x) = a*x^2 \) \{sym Associative\}

\( a*x^2 = a*x^2 \) \{cong (sym SqIsMult)\}
Refactoring Square and Multiply

(a) Before

```c
// before ladderisation
...
int epsilon(int x, int n) {
    return x * x % n;
}

int lambda(int x, int a, int n) {
    return x * a % n;
}

int squaringExp(int a, int k, int n) {
    int x=1, i;
    int y = lambda(x, a, n);
    for (i = MSB ; i >= 0 ; i--) {
        x = epsilon(x, n);
        if (bit(k,i) == 1) {
            x = lambda(x, a, n);
        }
    }
    return x;
}
```

(b) After

```c
// after ladderisation
...
int epsilon(int x, int n) {
    return x * x % n;
}

int lambda(int x, int a, int n) {
    return x * a % n;
}

int f(int x, int y, int n) {
    return x * y % n;
}

int squaringExp(int a, int k, int n) {
    int x=1, i;
    int y = lambda(x, a, n);
    for (i = MSB ; i >= 0 ; i--) {
        if (bit(k,i) == 1) {
            x = f(x, y, n);
            y = epsilon(y, n);
        } else {
            y = f(y, x, n);
            x = epsilon(x, n);
        }
    }
    return x;
}
```
Automatic ladderisation: improving code security through rewriting and dependent types

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 a program’s execution, e.g. timing or power leakage, to uncover secret information, such as passwords or sensitive data. There has been various attempts at addressing the problem of side-channel attacks, often relying

1 INTRODUCTION
Writing applications with security in mind is often a neglected activity amongst developers, where the focus on development has been typically focussed on timing optimisations. Indeed, security attacks on software and devices are becoming more commonplace as attackers exploit vulnerabilities in the underlying 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

• Currently in preparation
• Jointly with INRIA and Imperial
• Planned submission to PEPM 2021
Summary

• Security is a neglected goal amongst developers

• Writing code with security in mind if challenging for the average developer

• A full tool-supported methodology
  • Refactoring, rewriting, soundness,

• (Semi-)automatically transform a program into it’s secure (semi interleaved) version
Future Work

• Results, use-cases showing security properties of refactored code
  • Currently ongoing
• Fully-interleaved ladders
• Multiple rewrite suggestions to consider... which one is the best?
• Other security properties
• Security “patterns”? 
Thank you!

cmb21@st-andrews.ac.uk

@chrismarkbrown