A Verified Information-Flow Architecture

Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hrițcu, David Pichardie, Benjamin Pierce, Randy Pollack, Andrew Tolmach

January 2014
What if we could redesign computers for security?
Clean-slate redesign of entire system stack

http://www.crash-safe.org
The SAFE Project

- Support for critical security primitives from hardware to application levels
  - Memory safety
  - Strong dynamic typing
  - Dynamic information flow and access control
- Design of key components informed by verification
SAFE Architecture

- Integrate well-known mechanisms for security (e.g. fat-pointers, type tags, ...)

- Focus of this paper: flexible mechanism for supporting information-flow control (IFC) using a hardware cache and tags

```
payload  tag
```
Our Goal

Formalize and verify the core IFC mechanism proposed by SAFE using the Coq proof assistant
**Information-Flow Control (IFC)**

Track and limit information dependency in computations

**Noninterference (NI):** Varying secret inputs does not affect public observations
Concrete Machine

Core model of SAFE
Concrete Machine

Abstract Machine

Easier to show

Concrete Machine

Preserved

How can we prove it?

IFC baked into semantics

Refined by Core model of SAFE

Relies on determinism
Concrete Machine \[ \text{NI} \]

How can we prove it?
Abstract Machine

Refined by

IFC baked into semantics

Concrete Machine

NI
Abstract Machine

Refined by

Concrete Machine

NI

Easier to show

IFC baked into semantics

Core model of SAFE

Relies on determinism

How can we prove it?
Abstract Machine

Concrete Machine

Refined by

Preserved
Abstract Machine

Concrete Machine

Refined by

Preserved

Relies on determinism

NI
Our Contributions

Using Coq, we

- Model the core mechanism for supporting IFC in SAFE (our so-called “concrete machine”)

- Develop a proof methodology (by refinement) for proving this mechanism correct
Information-Flow Model
Abstract Machine

Refined by

Concrete Machine
Simple Abstract Machine

Stack machine with output channel, operating on integers
Simple Abstract Machine

Input/output model:

- The input of a program is its initial stack
- The result of executing a program is the sequence of its outputs
**IFC Abstract Machine**

Label data with security levels

H = high security  
L = low security

(or, more generally, any IFC lattice)
Machine has semantics with standard dynamic IFC baked-in

Mechanizing proof of NI is relatively straightforward

IFC Abstract Machine

Label data with security levels

- H = high security
- L = low security

(or, more generally, any IFC lattice)

When low-security operands are combined, the result is low-security.

When one operand is high-security, result is high-security.

Labels on outputs mark who is able to see them.

···
IFC Abstract Machine

When low-security operands are combined, the result is low-security

- 1@L
- 5@L
- 8@H

... Add Add Output ...

PC
IFC Abstract Machine

When low-security operands are combined, the result is low-security

```
... Add Add Output ...
```

PC

```
1@L
5@L
8@H
```
IFC Abstract Machine

When low-security operands are combined, the result is low-security

\[ \cdots \text{Add} \text{Add} \text{Output} \cdots \]

PC

6@L

8@H
IFC Abstract Machine

When one operand is high-security, result is high-security

... Add Add Output ...

PC

6@L
8@H
IFC Abstract Machine

When one operand is high-security, result is high-security

\[ \ldots \text{Add} \text{Add} \text{Output} \ldots \]

PC
When one operand is high-security, result is high-security

\[ \cdots \text{Add} \text{Add} \text{Output} \cdots \]
IFC Abstract Machine

Labels on outputs mark who is able to see them.

... Add Add Output ...

PC

14@H
IFC Abstract Machine

Labels on outputs mark who is able to see them

... Add Add Output ...

PC

14@H

H = high security
L = low security
(or, more generally, any IFC lattice)

Machine has semantics with standard dynamic IFC baked-in

Mechanizing proof of NI is relatively straightforward

When low-security operands are combined, the result is low-security
When one operand is high-security, result is high-security

Labels on outputs mark who is able to see them
IFC Abstract Machine

Labels on outputs mark who is able to see them

... Add Add Output ...

PC

14@H

(or, more generally, any IFC lattice)

Mechanizing proof of NI is relatively straightforward
IFC Abstract Machine

- Label data with security levels:
  - H = high security
  - L = low security
  - (or, more generally, any IFC lattice)

- Machine has semantics with standard dynamic IFC baked-in

- Mechanizing proof of NI is relatively straightforward

- When low-security operands are combined, the result is low-security.

- When one operand is high-security, the result is high-security.

- Labels on outputs mark who is able to see them

---

```
... Add Add Output ...
```

---

PC

---

14@H
IFC in Hardware
Abstract Machine

Concrete Machine
Concrete Machine

Similar to previous one, but with **hardware mechanisms** for supporting IFC
Concrete Machine

Plain integer tags instead of high-level IFC labels

Uninterpreted in hardware
Concrete Machine

Hardware *cache* governs tag propagation

![Diagram of a concrete machine with cache tags](image-url)
Concrete Machine

Cache line relates combination of instruction and operand tags to result tag.
Cache Operation

```
Add 0 0 ?
```

Cache:

```
1@0
5@0
8@1
18
```
Cache Operation

Result is 0

Control is transferred to fault handler
Cache Operation

Add 0 0 ?

Result is 0

Cache

Add 0 0
Cache Operation

Result is 0

Control is transferred to fault handler
Cache Operation

Add

Result is 0

Add

miss!

Control is transferred to fault handler
Cache Operation

Add 0 1

Cache

6@0
8@1

Result is 0

miss!

Control is transferred to fault handler
Cache Operation

6@0
8@1

Add 0 1

Result is 0

Cache

miss!
Cache Operation

Add 0 1

6@0
8@1

miss!

Control is transferred to fault handler

Result is 0
Cache Operation

Control is transferred to fault handler
Fault-Handler Operation

Handler is a piece of machine code running in \textit{privileged mode}.
Fault-Handler Operation

It can modify the cache, as well as bypass it.
Fault-Handler Operation

It analyzes the fault context

Fault Handler

Add 0 1 ?
Fault-Handler Operation

It analyzes the fault context.
Fault-Handler Operation

It analyzes the **fault context**
Fault-Handler Operation

It analyzes the fault context.
Fault-Handler Operation

... and computes corresponding result tag
Fault-Handler Operation

The handler then installs that line in the cache, returning to faulting instruction.
Fault-Handler Operation

Cache look-up will then succeed, allowing code to continue
Fault-Handler Operation

Fault Handler is a piece of machine code running in privileged mode. It can modify the cache, as well as bypass it. It analyzes the fault context and computes corresponding result tag. The handler then installs that line in the cache, returning to the faulting instruction.

Cache look-up will then succeed, allowing code to continue.

![Diagram showing cache look-up success]

- Result is 1
Fault-Handler Operation

Cache look-up will then succeed, allowing code to continue.
Proving the Refinement
Abstract Machine

Refined by

Concrete Machine
Refinement Structure

Abstract

$a_1$

labels correctly represented, cache compatible with IFC rules, machine in user mode

$c_1$

Concrete
Refinement Structure

Abstract

machine runs, producing output trace

Concrete

$\text{Concrete}$

$\text{Abstract}$
Refinement Structure

want corresponding abstract trace

combine two execution lemmas

Abstract

Concrete
Refinement Structure

Abstract

Cache hit step

Concrete

\(a_1\)

\(c_1\) \(\rightarrow c_2\)

\(t_a\)

\(t_c\)
Refinement Structure

Abstract

hit case simulation lemma

Concrete
Refinement Structure

Abstract

Concrete

cache miss step

$\mathbf{a}_1 \rightarrow \mathbf{a}_2 \rightarrow \ldots \rightarrow t_a$

$\mathbf{c}_1 \rightarrow \mathbf{c}_2 \rightarrow \mathbf{p}_1 \rightarrow \ldots \rightarrow \mathbf{p}_k \rightarrow \mathbf{c}_3 \rightarrow \ldots$

$\mathbf{o}_a \rightarrow \mathbf{o}_c$

$t_c$
Refinement Structure

Abstract

handler executes ...

Concrete
Refinement Structure

... and returns to user code

Concrete

Abstract

a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow t_a

\text{Abstract}

c_1 \rightarrow c_2 \rightarrow p_1 \rightarrow \cdots \rightarrow p_k \rightarrow c_3 \rightarrow \cdots

\text{Concrete}

o_a \quad o_c
Refinement Structure

Concrete

Abstract

handler correctness lemma

22
Refinement Structure

Abstract

Concrete

proceed inductively

\[ c_1 \rightarrow c_2 \rightarrow p_1 \rightarrow \ldots \rightarrow p_k \rightarrow c_3 \rightarrow \ldots \]
Challenges

Interesting issues involved in these proofs

- Verification of machine code is difficult
- Need to formalize notion of compatibility between cache and IFC rules
- How to make it work for any IFC lattice?
Idea: Structure proof and implementation with another refinement
Abstract Machine

Symbolic-Rule Machine

Concrete Machine

Fault Handler

Refined by

Refined by

IFC Rules

IFC side-conditions symbolically represented by
Abstract Machine

Symbolic-Rule Machine

Concrete Machine

IFC Rules

Fault Handler

Refined by

IFC side-conditions symbolically represented by
Abstract Machine

Symbolic-Rule Machine

Concrete Machine

IFC side-conditions symbolically represented by

Easy to show

Refined by

Refined by

IFC Rules

Fault Handler
Abstract Machine

Symbolic-Rule Machine

Concrete Machine

IFC side-conditions symbolically represented by

Refined by

Provide formulation of cache compatibility

Refined by

Fault Handler
Abstract Machine

Symbolic-Rule Machine

Concrete Machine

IFC Rules

Fault Handler

IFC side-conditions symbolically represented by

Correctly compiled to

Refined by

Refined by
Abstract Machine

Symbolic-Rule Machine

Concrete Machine

Refined by

IFC Rules

Refined by

Correctly compiled to

IFC side-conditions symbolically represented by

Combine hit and miss simulation lemmas
Symbolic Rule Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>LAB₁⊔LAB₂</td>
</tr>
<tr>
<td>Output</td>
<td>LAB₁</td>
</tr>
</tbody>
</table>

if the current instruction is
## Symbolic Rule Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>LAB₁ ⊔ LAB₂</td>
</tr>
<tr>
<td>Output</td>
<td>LAB₁</td>
</tr>
</tbody>
</table>

...  

label the result with
Symbolic Rule Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>LAB$_1$ \cup$LAB$_2$</td>
</tr>
<tr>
<td>Output</td>
<td>LAB$_2$</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

for Add, result is as secret as operands
## Symbolic Rule Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>LAB₁⊔LAB₂</td>
</tr>
<tr>
<td>Output</td>
<td>LAB₁</td>
</tr>
</tbody>
</table>

for Output, use same label
Handler
Implementation and Verification
Structured-code Generators

Structured programming instead of assembly programming

- Define structured-code generators as Coq functions
- Generators provide a structured language for the machine (if, case, and, or, while, ...)
- Prove Hoare-logic rules for each generator
Compiling IFC Rules

Write a rule table compiler in Coq

- Use generators as a backend
- Parameterized over correct implementation of lattice primitives
- Compose Hoare triples to show compiler correctness
Algorithm

- Fetch instruction and operand tags from faulting context
- Compute the result tag from this data using compiled rule table
- **Install** computed line into the cache

Proven correct by composing compiler lemma with triples for the glue code
Abstract Machine

Symbolic-Rule Machine

Concrete Machine

IFC side-conditions symbolically represented by

Refined by

Correctly compiled to

Fault Handler

NI
What Else?
Complete model includes more features

- Control flow and user-level procedures
- Block-structured memory with dynamic allocation
- System calls for implementing new IFC primitives
- Richer IFC labels (sets of principals represented as pointers to memory arrays)
Addressed Challenges

- Track implicit flows
Addressed Challenges

- Track implicit flows
- Allocation and noninterference
Addressed Challenges

- Track implicit flows

- Allocation and noninterference
  - Pointer values could leak secrets
Addressed Challenges

- Track implicit flows
- Allocation and noninterference
  - Pointer values could leak secrets
- Label representation depending on machine state
Addressed Challenges

- Track implicit flows
- Allocation and noninterference
  - Pointer values could leak secrets
- Label representation depending on machine state
- Low-level code for array manipulation and corresponding proofs
Wrapping Up
Conclusions

- Described a hardware mechanism for dynamic tag-checking and propagation
- Proof architecture for connecting it to high-level property
  - Refinement provides structure to proof
- Everything formalized in Coq
Coq Development

Entire formalization available at www.crash-safe.org
Complete machine and corresponding proofs in approximately 15k LOC
Future Work

- More interesting IFC features (concurrency, declassification, dynamic principal generation, . . . )

- Make model more realistic
  - Incorporate more features of SAFE
  - Study tag cache in the context of conventional architectures

- Mechanism is not IFC-specific, investigate other applications.
Thank You!

Questions?