CRASH/SAFE: Clean-slate Co-design of a Secure Host Architecture

Cătălin Hrițcu
Outline

• Overview of CRASH/SAFE project
  – clean-slate co-design of a secure host architecture

• Exceptions and information flow control (IFC)
  – to appear at IEEE S&P 2013 (Oakland)

• Testing noninterference with QuickCheck
  – ready for ICFP 2013 (deadline in 24 hours)

• Future directions
CRASH/SAFE project

• Academic partners (16):
  – University of Pennsylvania (11)
  – Harvard University (4)
  – Northeastern University (1)

• Industrial partners (24):
  – BAE systems (21) + Clozure (3)

• Funded by DARPA
  – Clean-Slate Design of Resilient, Adaptive, Secure Hosts
Clean-slate co-design of net host
Clean-slate co-design of net host

Primary goal:
design and implement a significantly more secure architecture, without backwards compatibility concerns
Clean-slate co-design of net host

Primary goal:
design and implement a significantly more secure architecture, without backwards compatibility concerns

New stack:
- language
- system
- hardware
Clean-slate co-design of net host

Primary goal: design and implement a significantly more secure architecture, without backwards compatibility concerns

Secondary goal: verify that it’s secure (whatever that means)

New stack:
• language
• system ✔
• hardware
Grandpa! Why are computers so insecure?

Transistors were precious back then, my boy ...
Hardware is now abundant
Time for a redesign targeting security!

language

system

hardware
Time for a redesign targeting security!

language  cool ideas  system  hardware
Time for a redesign targeting security!

- language
- system
- hardware

cool ideas
Time for a redesign targeting security!

- language
- system
- hardware

- Information flow
- Access control
- Type safety
- Memory safety
Time for a redesign targeting security!

- Language system
- Information flow
- Access control
- Type safety
- Memory safety

Fine-grained protection
Basic abstraction
Time for a redesign targeting security!

- Language
- System
- Hardware

- Information flow
- Access control
- Type safety
- Memory safety

- Fine-grained protection
- Basic abstraction

Verification
Language (Breeze)

• testing ground for ideas we port to lower levels
• **type and memory safe** high-level language
  – dynamically typed + dynamically-checked contracts
• **functional core** ($\lambda$) + state(!) + concurrency ($\pi$)
  – message-passing communication (channels)
Language (Breeze)

• testing ground for ideas we port to lower levels
• **type and memory safe** high-level language
  – dynamically typed + dynamically-checked contracts
• **functional core** \((\lambda)\) + state(\(!\)) + concurrency \((\Pi)\)
  – message-passing communication (channels)
• **built-in fine-grained protection mechanisms:**
  – values are attached **security labels** (e.g. public/secret)
  – **dynamic information flow control** (IFC)
  – **discretionary access control** (clearance)
Language (Breeze)

- testing ground for ideas we port to lower levels
- **type and memory safe** high-level language
  - dynamically typed + dynamically-checked contracts
- **functional core** ($\lambda$) + state(!) + concurrency ($\Pi$)
  - message-passing communication (channels)
- built-in **fine-grained protection mechanisms**:
  - values are attached **security labels** (e.g. public/secret)
  - dynamic information flow control (IFC)
  - discretionary access control (clearance)
- novel **exception handling mechanism** (more later)
Runtime/operating system

• manages:
  – **time** - scheduler
  – **memory** - allocator, garbage collector
  – **communication and devices** - channels
  – **protection** – dynamic IFC and access control
Runtime/operating system

• manages:
  – time - scheduler
  – memory - allocator, garbage collector
  – communication and devices - channels
  – protection – dynamic IFC and access control

• zero-kernel operating system
  – reduced TCB even wrt microkernel
  – least privilege & privilege separation taken to extreme
  – kernel split into mutually distrustful federated services
Hardware

• all instructions have well-defined semantics
  – abstractions strictly enforced
Hardware

• all instructions have well-defined semantics
  – abstractions strictly enforced
• low-fat pointers
  – can’t access/write out of frame bounds
Hardware

• all instructions have well-defined semantics
  – abstractions strictly enforced
• low-fat pointers
  – can’t access/write out of frame bounds
• dynamic types
  – can’t turn ints into pointers (unforgeable capabilities)
Hardware

• all instructions have well-defined semantics
  – abstractions strictly enforced
• low-fat pointers
  – can’t access/write out of frame bounds
• dynamic types
  – can’t turn ints into pointers (unforgeable capabilities)
• closures ($\lambda$) + protected call stack
Hardware

• all instructions have well-defined semantics
  – abstractions strictly enforced
• low-fat pointers
  – can’t access/write out of frame bounds
• dynamic types
  – can’t turn ints into pointers (unforgeable capabilities)
• closures ($\lambda$) + protected call stack
• authority + gates (authority switching closures)
  – fine-grained privilege separation / cheap system calls
Hardware

- all instructions have well-defined semantics
  - abstractions strictly enforced
- low-fat pointers
  - can’t access/write out of frame bounds
- dynamic types
  - can’t turn ints into pointers (unforgeable capabilities)
- closures ($\lambda$) + protected call stack
- authority + gates (authority switching closures)
  - fine-grained privilege separation / cheap system calls
- programmable tag management unit (TMU)
Tag management

• every word tagged with arbitrary pointer
  – only operating system interprets these pointers

• on each instruction TMU looks up tags of operands in a hardware rule cache
  – found $\rightarrow$ rule provides tags on results (no delay)
  – not found $\rightarrow$ trap to software (protection server)

• extremely fine-grained access control + dynamic IFC enforced at the lowest level
Robust Exception Handling for Sound Fine-Grained Dynamic Dynamic IFC

Cătălin Hrițcu, Michael Greenberg, Ben Karel, Benjamin Pierce, Greg Morrisett

IEEE Symposium on Security & Privacy 2013 (Oakland)
Exception handling

• we wanted reliable error recovery in Breeze
  – recovery from all exceptions including IFC violations
• however, existing work assumes errors are **fatal**
Exception handling

• we wanted reliable error recovery in Breeze
  – recovery from all exceptions including IFC violations
• however, existing work assumes errors are fatal
  – makes some things easier ... at the expense of others
    +secrecy  +integrity  –availability
Problem #1: IFC exceptions reveal information about labels

- labels are themselves information channels
- get soundness by preventing secrets from leaking either into or out of label channel
Problem #1: IFC exceptions reveal information about labels

• labels are themselves information channels
• get soundness by preventing secrets from leaking either *into* or *out of* label channel

allow labels to depend on secrets

labels must be hidden
Problem #1: IFC exceptions reveal information about labels

- labels are themselves information channels
- get soundness by preventing secrets from leaking either into or out of label channel
- secret bit: s@secret \(\text{low} \leq \text{secret} \leq \text{top-secret}\)
Problem #1: IFC exceptions reveal information about labels

• labels are themselves information channels
• get soundness by preventing secrets from leaking either into or out of label channel
• secret bit: $s@secret$  $\text{low} \leq \text{secret} \leq \text{top-secret}$

\[
\text{encode s into label} \\
\begin{align*}
&(\text{if } s \text{ then } ()@secret \\
&\text{else } ()@\text{top-secret});
\end{align*}
\]
Problem #1: IFC exceptions reveal information about labels

• labels are themselves information channels
• get soundness by preventing secrets from leaking either into or out of label channel

• secret bit: s@secret

\[
\text{l@low} \leq \text{secret} \leq \text{top-secret}
\]

let href = ref secret () in

......

\[
\text{href} := \begin{cases} 
\text{if s then } ()@\text{secret} & \text{if branch - assignment works} \\
\text{else } ()@\text{top-secret} & \text{else branch - IFCException}
\end{cases}
\]
Problem #1: IFC exceptions reveal information about labels

• labels are themselves information channels

• get soundness by preventing secrets from leaking either into or out of label channel

• secret bit: $s\leqslant secret \leqslant top-secret$

```haskell
let href = ref secret () in .......
try
  href := (if $s$ then ()@secret
          else ()@top-secret);
  true
catch IFCException => false
```
Problem #1: IFC exceptions reveal information about labels

- labels are themselves information channels
- get soundness by preventing secrets from leaking either into or out of label channel

allow labels to depend on secrets

labels must be hidden

IFC errors must be hidden! (and we don’t want that)

```latex
if s then ()@secret else ()@top-secret
```
Problem #1: IFC exceptions reveal information about labels

- labels are themselves information channels
- get soundness by preventing secrets from leaking either \underline{into} or \underline{out of} label channel

enforce that labels don’t depend on secrets

labels and IFC errors can be observed (“public labels”)

```
if s then ()@secret else ()@top-secret
```
Problem #1: IFC exceptions reveal information about labels

- labels are themselves information channels
- get soundness by preventing secrets from leaking either into or out of label channel

enforce that labels don’t depend on secrets

labels and IFC errors can be observed ("public labels")

Solution #1: brackets

```latex
\text{top-secret}[\text{if } s \text{ then } ()@\text{secret} \text{ else } ()@\text{top-secret}]
```
Problem #2: exceptions destroy control flow join points

• ending brackets need to be control flow join points, otherwise...
  
  - try
    
    let _ = secret[if h then throw Ex] in
    false
    catch Ex => true
Problem #2: exceptions destroy control flow join points

- ending brackets need to be control flow join points, otherwise...
  - try
    let _ = secret[if h then throw Ex] in
    false
    catch Ex => true

- brackets need to delay all exceptions!
  - secret[if true@secret then throw Ex] => "(Error Ex)@secret"
  - secret[if false@secret then throw Ex] => "(Success ())@secret"
Problem #2: exceptions destroy control flow join points

• ending brackets need to be control flow join points, otherwise...
  – try
    let _ = secret[if h then throw Ex] in
    false
    catch Ex => true

• brackets need to delay all exceptions!
  – secret[if true@secret then throw Ex] => “(Error Ex)@secret”
  – secret[if false@secret then throw Ex] => “(Success ())@secret”

• similarly for failed brackets
  – secret[42@top-secret] => “(Error EBracket)@secret”
Solution #2: Delayed exceptions

- delayed exceptions unavoidable
  – still have a choice how to propagate them

- we studied two main alternatives:
  1. mix active and delayed exceptions \((\lambda ^1 \text{throw})\)
Solution #2: Delayed exceptions

• delayed exceptions unavoidable
  – still have a choice how to propagate them

• we studied two main alternatives:
  1. mix active and delayed exceptions \( (\lambda[^1] \text{throw}) \)
  2. only delayed exceptions \( (\lambda[^1]_{\text{NaV}}) \)

  • delayed exception = not-a-value (NaV)
  • NaVs are first-class replacement for values
  • NaVs propagated solely via data flow
  • NaVs are labeled and pervasive
  • simpler and more radical solution; implemented in Breeze
What’s in a NaV? Debugging aids!

• error message
  – `EDivisionByZero (“can’t divide %1 by 0”, 42)

• stack trace
  – pinpoints error origin
    (not the billion-dollar mistake!)

• propagation trace
  – how did the error make it here?
What’s in a NaV? Debugging aids!

• error message
  – `EDivisionByZero (“can’t divide %1 by 0”, 42)

• stack trace
  – pinpoints error origin
    (not the billion-dollar mistake!)

• propagation trace
  – how did the error make it here?

NaVs are compiler writer’s dream, especially if compiler is allowed to be imprecise about these debugging aids (Greg Morrisett)
NaV-lax vs. NaV-strict behavior

- all non-parametric operations are NaV-strict
  - NaV@low + 42@high => NaV@high

- for parametric operations we can chose:
  NaV-lax or NaV-strict
  - (fun x => 42) NaV => 42 or => NaV
NaV-lax vs. NaV-strict behavior

• all non-parametric operations are NaV-strict
  – NaV@low + 42@high => NaV@high

• for parametric operations we can chose:
  NaV-lax or NaV-strict
  – (fun x => 42) NaV => 42 or => NaV
  – Cons NaV Nil => Cons NaV Nil or => NaV
NaV-lax vs. NaV-strict behavior

• all non-parametric operations are NaV-strict
  – NaV@low + 42@high => NaV@high

• for parametric operations we can chose:

  NaV-lax          or          NaV-strict

  – (fun x => 42) NaV => 42          or          => NaV
  – Cons NaV Nil => Cons NaV Nil     or          => NaV
  – (r := NaV,r=7) => ((),r=NaV)    or          => (NaV,r=7)
NaV-lax vs. NaV-strict behavior

• all non-parametric operations are NaV-strict
  – NaV@low + 42@high => NaV@high

• for parametric operations we can chose:
  \[
  \begin{align*}
  \text{NaV-lax} & \quad \text{or} \quad \text{NaV-strict} \\
  (\text{fun } x \Rightarrow 42) \text{ NaV} & \Rightarrow 42 \quad \text{or} \quad \Rightarrow \text{NaV} \\
  \text{Cons NaV Nil} & \Rightarrow \text{Cons NaV Nil} \quad \text{or} \quad \Rightarrow \text{NaV} \\
  (r := \text{NaV}, r=7) & \Rightarrow ((), r=\text{NaV}) \quad \text{or} \quad \Rightarrow (\text{NaV}, r=7)
  \end{align*}
  \]

• NaV-strict behavior reveals errors earlier
  – but it also introduces additional IFC constraints
  – applied everywhere it makes brackets useless

• in Breeze the programmer can choose
  – in formal development NaV-lax everywhere
Formal results

• proved termination-insensitive noninterference in Coq for \( \lambda[] \), \( \lambda[]_{\text{NaV}} \), and \( \lambda[]_{\text{throw}} \)
  – for \( \lambda[]_{\text{NaV}} \) even with all debugging aids; error-sensitive

• in our setting NaVs and catchable exceptions have equivalent expressive power
  – translations validated by QuickChecking extracted code
Summary for IFC exceptions

• reliable error handling possible even for sound fine-grained dynamic IFC systems
• two mechanisms ($\lambda^_\text{NaV}$ and $\lambda^_\text{throw}$)
  – all errors recoverable, even IFC violations
  – necessary ingredients: sound public labels (brackets) + delayed exceptions
  – quite radical design (not backwards compatible!)
  – we believe delayed exceptions applicable to static IFC
Testing Noninterference, Quickly

Cătălin Hrițcu, John Hughes, Benjamin C. Pierce,
Antal Spector-Zabusky, Dimitrios Vytiniotis,
Arthur Azevedo de Amorim, Leonidas Lampropoulos

ready for submission to
International Conference on Functional Programming (ICFP 2013)
protection server

• most security-critical & novel component of our system
  – best target for verification

machine running protection server code noninterference (security)
protection server

• most security-critical & novel component of our system
  – best target for verification

more abstract machine with built-in IFC (executable spec)

more concrete machine running protection server code

noninterference (security)
protection server

- most security-critical & novel component of our system
  - best target for verification

- more abstract machine with built-in IFC (executable spec)
- more concrete machine running protection server code

noninterference (security)

correctness of implementation
protection server

• most security-critical & novel component of our system
  – best target for verification

Can we QuickCheck this?

more abstract machine with built-in IFC (executable spec)

more concrete machine running protection server code

correctness of implementation

noninterference (security)
Yes we can!

• random testing noninterference of pico-machine
  – simple stack machine with dynamic IFC (10 instrs.)
    • Push, Load, Store, Add, Sub, Noop, Jump, Call, Ret, Halt
Yes we can!

• random testing noninterference of pico-machine
  – simple stack machine with dynamic IFC (10 instrs.)
    • Push, Load, Store, Add, Sub, Noop, Jump, Call, Ret, Halt
  – designing sound IFC mechanism still tricky!
    • Jump / Call – to secret address raises PC label
    • Ret – unwinds stack, taints result, restores PC label
    • Store – no-sensitive-upgrade check [Austin&Flanagan, ‘09]:
      pc-label `join` label-of-address <= label-of-value-stored-at-address
Yes we can!

• random testing noninterference of pico-machine
  – simple stack machine with dynamic IFC (10 instrs.)
    • Push, Load, Store, Add, Sub, Noop, Jump, Call, Ret, Halt
  – designing sound IFC mechanism still tricky!
    • Jump / Call – to secret address raises PC label
    • Ret – unwinds stack, taints result, restores PC label
    • Store – no-sensitive-upgrade check [Austin&Flanagan, ‘09]:
      pc-label `join` label-of-address <= label-of-value-stored-at-address
• we proved noninterference for this in Coq in 1 week!
  why bother with testing?
Yes we can!

- random testing noninterference of pico-machine
  - simple stack machine with dynamic IFC (10 instrs.)
    - Push, Load, Store, Add, Sub, Noop, Jump, Call, Ret, Halt
  - designing sound IFC mechanism still tricky!
    - Jump / Call – to secret address raises PC label
    - Ret – unwinds stack, taints result, restores PC label
    - Store – no-sensitive-upgrade check [Austin&Flanagan, ‘09]:
      pc-label `join` label-of-address <= label-of-value-stored-at-address

- we proved noninterference for this in Coq in 1 week!
  why bother with testing?
  - we hope that QuickCheck will scale better than Coq to the
    much more complicated real SAFE machine (~110 instrs.)
How do we do it?

- Clever program generation strategies

<table>
<thead>
<tr>
<th>gen. strategy</th>
<th># bugs found</th>
<th>mean time to find</th>
<th>max time to find</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>4 out of 6</td>
<td>3030.30ms</td>
<td>&gt; 300s</td>
</tr>
<tr>
<td>weighted</td>
<td>4 out of 6</td>
<td>201.20ms</td>
<td>&gt; 300s</td>
</tr>
<tr>
<td>+ sequences</td>
<td>6 out of 6</td>
<td>16.45ms</td>
<td>300s</td>
</tr>
<tr>
<td>+ smart integers</td>
<td>6 out of 6</td>
<td>5.85ms</td>
<td>16.66s</td>
</tr>
<tr>
<td>+ gen. by exec.</td>
<td>6 out of 6</td>
<td>1.51ms</td>
<td>1.52s</td>
</tr>
</tbody>
</table>
How do we do it?

- Clever program generation strategies

<table>
<thead>
<tr>
<th>gen. strategy</th>
<th># bugs found</th>
<th>mean time to find</th>
<th>max time to find</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>4 out of 6</td>
<td>3030.30ms</td>
<td>&gt; 300s</td>
</tr>
<tr>
<td>weighted</td>
<td>4 out of 6</td>
<td>201.20ms</td>
<td>&gt; 300s</td>
</tr>
<tr>
<td>+ sequences</td>
<td>6 out of 6</td>
<td>16.45ms</td>
<td>300s</td>
</tr>
<tr>
<td>+ smart integers</td>
<td>6 out of 6</td>
<td>5.85ms</td>
<td>16.66s</td>
</tr>
<tr>
<td>+ gen. by exec.</td>
<td>6 out of 6</td>
<td>1.51ms</td>
<td>1.52s</td>
</tr>
</tbody>
</table>

- Shrinking counterexamples
How do we do it?

• Clever program generation strategies

<table>
<thead>
<tr>
<th>gen. strategy</th>
<th># bugs found</th>
<th>mean time to find</th>
<th>max time to find</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>4 out of 6</td>
<td>3030.30ms</td>
<td>&gt; 300s</td>
</tr>
<tr>
<td>weighted</td>
<td>4 out of 6</td>
<td>201.20ms</td>
<td>&gt; 300s</td>
</tr>
<tr>
<td>+ sequences</td>
<td>6 out of 6</td>
<td>16.45ms</td>
<td>300s</td>
</tr>
<tr>
<td>+ smart integers</td>
<td>6 out of 6</td>
<td>5.85ms</td>
<td>16.66s</td>
</tr>
<tr>
<td>+ gen. by exec.</td>
<td>6 out of 6</td>
<td>1.51ms</td>
<td>1.52s</td>
</tr>
</tbody>
</table>

• Shrinking counterexamples

• Stronger noninterference properties
noninterference

EENI  SNI

LLNI

SSNI
noninterference

what we actually want

for successfully terminating programs

EENI          SNI

LLNI

SSNI

for server loops
noninterference

what we actually want

EENI  SNI

for server loops

for successfully terminating programs

LLNI

what’s easy to test

SSNI
noninterference

what we actually want

for successfully terminating programs

EENI  SNI

for server loops

what’s easy to test

LLNI

what we can prove by (co)induction ("unwinding conditions")

SSNI
End-to-end noninterference (EENI)

what we actually want for terminating programs

differ only on secrets
Single-step noninterference (SSNI)

easy to test and suitable for proof ("unwinding conditions")

\[
\begin{align*}
L & \rightarrow* \rightarrow L \\
H & \rightarrow L \\
& \rightarrow \text{halt} \\
& \rightarrow* \\
\end{align*}
\]
Experiments

• Stronger properties discover bugs much faster

<table>
<thead>
<tr>
<th>strategy</th>
<th># bugs found</th>
<th>mean time to find</th>
<th>max time to find</th>
</tr>
</thead>
<tbody>
<tr>
<td>EENI + GenByExec</td>
<td>14 out of 14</td>
<td>549.45ms</td>
<td>300.00s</td>
</tr>
<tr>
<td>LLNI + GenByExec</td>
<td>14 out of 14</td>
<td>17.13ms</td>
<td>0.90s</td>
</tr>
<tr>
<td>SSNI + Naive</td>
<td>14 out of 14</td>
<td>26.70ms</td>
<td>0.45s</td>
</tr>
<tr>
<td>SSNI + TinyStates</td>
<td>14 out of 14</td>
<td>4.68ms</td>
<td>0.03s</td>
</tr>
</tbody>
</table>

• SSNI is very cool, but ...
  – SSNI requires discovering stronger invariants
  – invariants of SAFE machine are very complicated
Ongoing work on CRASH/SAFE

• verifying simple protection server in Coq
  – micro-machine: hardware types, dynamic allocation, principal generation, public labels
  – joint with Benjamin Pierce, Delphine Demange, Andrew Tolmach

• protecting data integrity with signatures
  – meaning(lessness) of IFC endorsement; reviving trademarks [Moris ‘73]
  – beyond data abstraction (dynamic sealing): caching contracts

• fine-grained higher-order containment

• Breeze design paper

• Tag management unit (TMU) design paper

• implementing Breeze labels cryptographically
Future directions

• Formally verified privacy-preserving distributed applications (e.g. ones based on zero-knowledge proofs)

protecting personal information + digital credentials → privacy-enabled identity systems

“proving you are over 18 without revealing your age”
Future directions

- Formally verified privacy-preserving distributed applications (e.g. ones based on zero-knowledge proofs)

- Fine-grained access control and integrity protection for mobile devices

“proving you are over 18 without revealing your age”
THE END