Micro-Policies

Formally Verified, Tag-Based Security Monitors

Cătălin Hrițcu

Inria Paris-Rocquencourt, Prosecco team
Micro-Policies collaborators

- **Formal methods & hardware architecture**

- **Current team:**
  - *UPenn*: Arthur Azevedo de Amorim, André DeHon, Benjamin Pierce, Antal Spector-Zabusky, Udit Dhawan
  - *Inria Paris*:
    - Cătălin Hrițcu, Yannis Juglaret
  - *Portland State*: Andrew Tolmach

- **Past**: DARPA CRASH/SAFE project (2011-2014)
Computer systems are insecure
Computer systems are insecure

• **Today’s computers are mindless bureaucrats**
  – “write past the end of this buffer” ... *yes boss!*
  – “jump to this untrusted integer” ... *right boss!*
  – “return into the middle of this instruction” ... *sure boss!*

• **Software bears most of the burden for security**
  – pervasive security enforcement impractical
  – bad security-performance tradeoff
  – just write secure code ... all of it!

• **Consequence:** vulnerabilities in every system
  – violations of well-studied safety and security policies
Micro-policies

- add **large tag** to each machine word

- words in memory and registers are all tagged

*Conceptual model, the hardware implements this efficiently (a bit more later)*
Tag-based instruction-level monitoring

```
<table>
<thead>
<tr>
<th>pc</th>
<th>tpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>tr0</td>
</tr>
<tr>
<td>r1</td>
<td>tr1</td>
</tr>
<tr>
<td>r2</td>
<td>tr2</td>
</tr>
</tbody>
</table>
```

```
| mem[0] | tm0 |
| mem[1] | tm1 |
| mem[2] | tm2 |
| mem[3] | tm3 |
```

decode(mem[1]) = add r0 r1 r2

```
| tpc | tr0 | tr1 | tr2 | tm1 |
```

```
add
```

```
monitor
```

```
allow
tpc’
tr0’
```
Tag-based instruction-level monitoring

\[
\begin{array}{|c|c|}
\hline
pc & tpc \\
\hline
r0 & tr0 \\
\hline
r1 & tr1 \\
\hline
r2 & tr2 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
mem[0] & tm0 \\
\hline
mem[1] & tm1 \\
\hline
mem[2] & tm2 \\
\hline
mem[3] & tm3 \\
\hline
\end{array}
\]

\[
\text{decode}(\text{mem}[1]) = \text{store} \ r0 \ r1
\]

\[
\begin{array}{cccccc}
\text{tpc} & \text{tr0} & \text{tr1} & \text{tm3} & \text{tm2} \\
\hline
\end{array}
\]

\[
\text{store}
\]

\[
\text{monitor}
\]

\[
\text{bad action stopped!}
\]

\[
\text{disallow}
\]
Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
- **expressive**: can enforce large number of policies
- **flexible**: tags and monitor defined by software
- **efficient**: accelerated using hardware caching
- **secure**: formally verified to provide security
Expressiveness

• Micro-policy mechanism can efficiently enforce:
  – memory safety
  – code-data separation
  – control-flow integrity
  – compartment isolation
  – taint tracking
  – information flow control
  – monitor self-protection
  – dynamic sealing

... and a lot more!

History:
• SAFE machine had separate HW mechanisms for many of these
• micro-policies were only used for IFC [Oakland’13, POPL’14]
• ... we only realized later how expressive they are [ASPLOS’15, Oakland’15]
Flexibility by example: memory safety

• Our memory safety micro-policy prevents
  – **spatial violations**: reading/writing out of bounds
  – **temporal violations**: use after free, invalid free
  – for **heap-allocated data** (for simplicity)

• Pointers become **unforgeable capabilities**
  – can only obtain a valid pointer to a memory region
    • by allocating that region or
    • by copying/offsetting an existing pointer to that region
Memory safety micro-policy

\[ p \leftarrow \text{malloc } k \]

fresh \( c \)  
(e.g. ++\( c \))

\[ p = A8F0 \@ \text{ptr}(c) \]
\[ q \leftarrow p + k \]
\[ c = c \]
\[ !p \leftarrow 7 \]

free \( p \)

\[ T_v ::= i \mid \text{ptr}(c) \]
\[ \text{tags on values} \]
\[ T_m ::= M(c,T_v) \mid F \]
\[ \text{tags on memory} \]

out of bounds
Memory safety micro-policy

\[ p = A8F0@\text{ptr}(c) \]

Free \( p \)

Use after free

\[ \neg c' = c \]

Out of bounds

\[ q \leftarrow p + k \]

\[ A8FK@\text{ptr}(c) = q \]

\[ !q < 42 \]

\[ T_v ::= i \mid \text{ptr}(c) \] tags on values

\[ T_m ::= M(c, T_v) \mid F \] tags on memory
Is it fast?

EFFICIENTLY EXECUTING MICRO-POLICIES
Efficiently executing micro-policies

<table>
<thead>
<tr>
<th>op</th>
<th>tpc</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>tci</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>tpc</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>tci</td>
</tr>
<tr>
<td>op</td>
<td>tpc</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>tci</td>
</tr>
<tr>
<td>op</td>
<td>tpc</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>tci</td>
</tr>
<tr>
<td>op</td>
<td>tpc</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>tci</td>
</tr>
</tbody>
</table>

lookup → zero overhead hits!

hardware cache

<table>
<thead>
<tr>
<th>tpc’</th>
<th>tr</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpc’</td>
<td>tr</td>
</tr>
<tr>
<td>tpc’</td>
<td>tr</td>
</tr>
<tr>
<td>tpc’</td>
<td>tr</td>
</tr>
</tbody>
</table>
Efficiently executing micro-policies

<table>
<thead>
<tr>
<th>op</th>
<th>tpc</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>tci</th>
<th>tpc'</th>
<th>tr</th>
</tr>
</thead>
</table>

lookup \[\downarrow\]
misses trap to software
produced "rule" cached

<table>
<thead>
<tr>
<th>op</th>
<th>tpc</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>tci</th>
<th>tpc'</th>
<th>tr</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>tpc</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>tci</td>
<td>tpc'</td>
<td>tr</td>
</tr>
<tr>
<td>op</td>
<td>tpc</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>tci</td>
<td>tpc'</td>
<td>tr</td>
</tr>
<tr>
<td>op</td>
<td>tpc</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>tci</td>
<td>tpc'</td>
<td>tr</td>
</tr>
<tr>
<td>op</td>
<td>tpc</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>tci</td>
<td>tpc'</td>
<td>tr</td>
</tr>
</tbody>
</table>

hardware cache
Simulations for **naive** implementation

memory safety + code-data separation + taint tracking + control-flow integrity

simple RISC processor: single-core 5-stage in-order Alpha

implemented on FPGA (part of SAFE machine) [FPGA ’13, TRETS ’15]
Targeted [micro-]architectural optimizations

- grouping opcodes and ignoring unused tags
  - increases effective rule cache capacity
- transferring only unique tags to/from DRAM
  - reduces runtime and energy overhead
- using much shorter tags for on-chip data caches
  - reduces runtime, energy, and area overhead
- caching composite policies separately
  - makes rule cache misses much cheaper

[ASPLOS’15]
Simulations for **optimized** implementation

memory safety + code-data separation + taint tracking + control-flow integrity

simple RISC processor: single-core 5-stage in-order Alpha

no free lunch

More details

[ASPLOS’15]
Is it secure?

FORMAL VERIFICATION IN COQ

[POPL’14, Oakland’15]
Memory safe abstract machine

Symbolic machine

Correctly implements

Concrete machine

Rule cache

Monitor

Micro-policy

Correctly implements

Memory safety micro-policy

Correctly implements*

ASM

*only proved for IFC [POPL 2014]
Abstract machine for $P$

Symbolic machine

Correctly implements

Concrete machine

Rule cache

Micro-policy

Monitor

Correctly implements

$P$ in $\{\text{IFC, CFI}\}$

Secure (e.g. noninterference)

Secure
Memory safety micro-policy

1. Sets of tags

- $T_v ::= i \mid \text{ptr}(c)$
- $T_m ::= M(c,T_v) \mid F$
- $T_{pc} ::= T_v$

2. Transfer function

- Record $\text{IVec} ::= \{ \text{op}:\text{opcode} ; t_{pc}:T_{pc} ; t_i:T_m ; ts: \ldots \}$
- Record $\text{OVec} (\text{op}:\text{opcode}) ::= \{ t_{rpc} : T_{pc} ; t_r : \ldots \}$
- transfer : (iv:IVec) -> option (OVec (op iv))
1. Sets of tags
\[ T_v ::= i \mid \text{ptr}(c) \]
\[ T_m ::= M(c,T_v) \mid F \]
\[ T_{pc} ::= T_v \]

2. Transfer function
Record \( \text{IVec} := \{ \text{op:opcode}; t_{pc}:T_{pc}; t_{i}:T_m; ts: \ldots \} \)
Record \( \text{OVec} (\text{op:opcode}) := \{ t_{rpc}:T_{pc}; t_{r}: \ldots \} \)

\text{transfer} : (iv:\text{IVec}) \to \text{option} (\text{O Vec} (\text{op iv}))

Definition \text{transfer} iv :=

match iv with
| \{ \text{op=Load}; t_{pc}=\text{ptr}(c_{pc}); t_{i}=M(c_{pc},i); ts=[\text{ptr}(c); M(c,T_v)]\}
  => \{ t_{rpc}=\text{ptr}(c_{pc}); t_{r}=T_v \}
| \{ \text{op=Store}; t_{pc}=\text{ptr}(c_{pc}); t_{i}=M(c_{pc},i); ts=[\text{ptr}(c); T_v; M(c,T_v')]\}
  => \{ t_{rpc}=\text{ptr}(c_{pc}); t_{r}=M(c,T_v) \}
...

...
Memory safety micro-policy

1. Sets of tags

\[ T_v ::= i \mid \text{ptr}(c) \]
\[ T_m ::= M(c, T_v) \mid F \]
\[ T_{pc} ::= T_v \]

2. Transfer function

Record \textbf{IVec} := \{ op:opcode \mid t_{pc}:T_{pc} \mid t_i:T_m \mid ts: \ldots \}

Record \textbf{OVect} (op:opcode) := \{ t_{rpc}:T_{pc} \mid t_r: \ldots \}

\textbf{transfer} : (iv:\textbf{IVec}) \rightarrow \text{option} (\textbf{OVect} (op \ iv))

3. Monitor services

Record \textbf{service} := \{ addr : \text{word}; sem : \text{state} \rightarrow \text{option} \text{state}; \ldots \}

Definition \textbf{mem_safety_services} : \text{list} service :=

[\textbf{malloc}; \textbf{free}; \textbf{base}; \textbf{size}; \textbf{eq}].

*This takes us beyond “noninterferent” reference monitors (more soon)*
Open problems

• Interaction with compiler, loader, linker, OS
• Secure micro-policy composition
• Reduced/more adaptive energy usage
• Modern RISC instruction set (e.g. ARM)
• More realistic processor
  (our-of-order execution, even multi-core)
Full abstraction

- **Golden standard** for secure compilation
  - $P \approx Q \iff \text{compile}(P) \approx \text{compile}(Q)$
  - $P \approx Q = \forall C. \ C[P] \text{ has the same behavior as } C[Q]$
  - *intuition*: low-level machine code contexts can’t do more harm than high-level contexts
  - can securely link compiled and untrusted machine code
- **Very strong, but rarely achieved in practice**
  - much stronger than compiler correctness
  - need a compiler & runtime that actually enforce high-level abstractions at the low level
  - *... and that’s currently too expensive!*
Targeting micro-policy machine

- **Micro-policies can efficiently protect abstractions**
- **Fully abstract compiler to micro-policy machine**
  - working on this with **Yannis Juglaret (FCS talk next week)**
  - Toy source language: Featherweight Java subset + updates
  - FJ classes protected from native classes they link with
  - Micro-policy combining: protects:
    - isolated compartments, entry points
    - linear return capabilities
    - dynamic typing
    - classes, methods
    - stack discipline
    - type safety
- **Long term goal**: functional programming language
Open problem: composition

• composing reference monitors is easy
  – ... as long as they can only stop execution

• richer interaction for micro-policies:
  – monitor services: malloc, free, ...
    classify, declassify, read label, ...

• secure micro-policy composition is difficult
  – e.g. composing anything with IFC can leak
  – memory safety + compartmentalization vs.
    compartmentalization + memory safety?
Take away

- **Micro-policies**, novel security mechanism
  - low level, fine grained, expressive, flexible, efficient, formally secure

- cool research direction with many interesting open problems for us and others to solve

- Thank you!