Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu
Inria Paris
Computers are insecure

- devastating low-level vulnerabilities
- programming languages, compilers, and hardware architectures
  - designed in an era of scarce hardware resources
  - too often trade off security for efficiency
- the world has changed (2016 vs 1972)
  - security matters, hardware resources abundant
  - time to revisit some tradeoffs
Hardware architectures

• **Today’s processors are mindless bureaucrats**
  – “write past the end of this buffer”
  – “jump to this untrusted integer”
  – “return into the middle of this instruction”

• **Software bears most of the burden for security**

• **Manufacturers have started looking for solutions**
  – 2015: Intel Memory Protection Extensions (MPX) and Intel Software Guard Extensions (SGX)
  – 2016: Oracle Silicon Secured Memory (SSM)

“Spending silicon to improve security”
Unsafe low-level languages

• C (1972) and C++ **undefined behavior**
  – including buffer overflows, checks too expensive
  – compilers optimize aggressively assuming undefined behavior will simply not happen

• **Programmers bear the burden for security**
  – just write secure code ... all of it

---

**[PATCH] CVE-2015-7547 --- glibc getaddrinfo() stack-based buffer overflow**

**DNS queries**

**vulnerable since May 2008**
Safer high-level languages

• memory safe (at a cost)

• useful abstractions for writing secure code:
  – GC, type abstraction, modules, immutability, ...

• not immune to low-level attacks
  – large runtime systems, in C++ for efficiency
  – unsafe interoperability with low-level code
    • libraries often have large parts written in C/C++
    • enforcing abstractions all the way down too expensive
Efficient Secure Compilation to Micro-Policies

2\textsuperscript{nd} part of this talk (more speculative)

1. Secure semantics for low-level languages
   • Formally: fully abstract compilation
     – holy grail, enforcing abstractions all the way down
     – currently this would be way too expensive

2. Secure interoperability with lower-level code
   • Key enabling technology: micro-policies
     – hardware-accelerated tag-based monitoring

1\textsuperscript{st} part of this talk
MICRO-POLICIES
Micro-Policies team

- **Formal methods & architecture & systems**

- **Current team:**
  - *Inria*: Cătălin Hrițcu, Yannis Juglaret
  - *UPenn*: Arthur Azevedo de Amorim, André DeHon, Benjamin Pierce, Nick Roessler, Antal Spector-Zabusky
  - *Portland State*: Andrew Tolmach
  - *MIT*: Howard E. Shrobe, Stelios Sidiropoglou-Douskos
  - *Industry*: Draper Labs, Bluespec Inc

- **Spinoff of past project:** DARPA CRASH/SAFE (2011-2014)
Micro-policies

• add large tag to each machine word (unbounded metadata)

<table>
<thead>
<tr>
<th>word</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  - tag[0]
  - tag[1]
  - tag[2]

(not addressable)

• words in memory and registers are all tagged

<table>
<thead>
<tr>
<th>pc</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>tag</td>
</tr>
<tr>
<td>r1</td>
<td>tag</td>
</tr>
<tr>
<td>r2</td>
<td>tag</td>
</tr>
</tbody>
</table>

| mem[0] | tag |
| mem[1] | tag |
| mem[2] | tag |
| mem[3] | tag |

*Conceptual model, our hardware implements this efficiently*
Tag-based instruction-level monitoring

decode(mem[1]) = add r0 r1 r2
Tag-based instruction-level monitoring

<table>
<thead>
<tr>
<th>pc</th>
<th>tpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>tpc</td>
</tr>
<tr>
<td>r1</td>
<td>tpc</td>
</tr>
<tr>
<td>r2</td>
<td>tpc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mem[0]</th>
<th>tm0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem[1]</td>
<td>tm1</td>
</tr>
<tr>
<td>mem[2]</td>
<td>tm2</td>
</tr>
<tr>
<td>mem[3]</td>
<td>tm3</td>
</tr>
</tbody>
</table>

decode(mem[1]) = store r0 r1

store

monitor

bad action stopped!
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**: simple enough to formally verify security
- **real**: FPGA implementation on top of RISC-V CPU
Expressiveness

- information flow control (IFC) [Oakland’13, POPL’14]
- monitor self-protection
- compartmentalization
- dynamic sealing
- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking
- ...

Verified (in Coq)
[Oakland’15]

Evaluated (<10% runtime overhead)
[ASPLOS’15]
Flexibility (by example)

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

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

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

\[ \text{fresh } c \] (e.g. ++c)

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

\[ q \leftarrow p + k \]

\[ c = c \]

\[ c \neq c' \]

\[ q \neq 42 \]

\[ \text{out of bounds} \]

\[ \text{free } p \]

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

\[ T_m ::= M(c,T_v) | F \] tags on memory
Memory safety micro-policy

\[ p = A8F0 \] @ptr(c) \]

\[ A8FK @ptr(c) = q \]

\[ q \leftarrow p + k \]

\[ c \neq c' \]

\[ \text{free } p \]

\[ x < !p \]

\[ \text{use after free} \]

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

\[ T_m ::= M(c, T_v) \mid F \]

Oracle Silicon Secured Memory (2016) similar, but with only 16 colors

Intel MPX cannot detect this
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>
</table>

lookup  
zero overhead hits!

found

<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>
</tbody>
</table>

hardware cache
Efficiently executing micro-policies

lookup

misses trap to software
produced “rule” cached

hardware cache
Experimental evaluation (simulations)

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

simple RISC processor: single-core 5-stage in-order Alpha (pre RISC-V transition)

spending silicon

% Area Ovhd

% Energy Ovhd

220% -> 60% (40% now)

50% -> 7%

More details [ASPLOS’15]
Formal verification in Coq

[POPL’14, Oakland’15]

Memory safe abstract machine

Symbolic machine

Concrete machine

Micro-policy

Monitor

Rule cache

Correctly implements

Correctly implements

Correctly implements* only proved for IFC (verified DSL compiler)

Generic Framework

ASM
Is this secure?

Concrete machine correctly implements Abstract machine for P which correctly implements Symbolic machine which correctly implements Micro-policy P in {IFC,CFI} secure (e.g. noninterference) secure

* Working on extrinsic definition of memory safety [Alpha is for address, Azevedo de Amorim et al, draft 2015]
SECURE COMPILATION

Joint work with Yannis Juglaret
Secure compilation

• **Goal:** to build the **first efficient** secure compilers for **realistic programming languages**

1. Secure semantics for low-level languages
   – C with memory safety and compartmentalization

2. Secure interoperability with lower-level code
   – ASM, C, ML, and F* (verification system for ML)
   – problems are quite different at different levels

• Formally: **fully abstract compilation**
  – enforcing abstractions all the way down
Benefits: can reason about security in the source language; forget about compiler, linker, loader, runtime system, and (to some extent) low-level libraries
Very long term vision

F*

SecF*

ML

SecML

C variants

memory safe

CompSec

CompSec

CompSec

CompSec

ASM (RISC-V+μP)

compiled F* component

compiled ML component

compiled safe C component

compiled legacy C comp

compiled legacy C comp

ASM component

compartmentalization boundaries
Low-level compartmentalization

- Break up software into **mutually distrustful components** running with **minimal privileges** & interacting only via **well-defined interfaces**
- **Limit the damage** of control hijacking attacks to just the C or ASM components where they occur
- Not a new idea, already deployed in practice:
  - process-level privilege separation
  - software-fault isolation
- Micro-policies can give us **better interaction model**
- We also aim to **show security formally**
Compartmentalized C

• Want to **add components with typed interfaces to C**

• Compiler (e.g. CompCert), linker, loader propagate interface information to low-level memory tags
  – each component’s memory tagged with unique color
  – procedure entry points tagged with procedure’s type

• **Micro-policy enforcing:**
  – component isolation
  – **procedure call discipline** (entry points)
  – **stack discipline for returns** (linear return capabilities)
  – **type safety** on cross-component interaction

Compartmentalization micro-policy

- **Jal r**
- **...**
- **@EntryPoint**
- **Store r_a \to \star r_m**
- **...**
- **Load \star r_m \to r_a**
- **Jump r_a**

**memory**

**registers**

\[ \text{pc} \quad \ldots \quad r \]

**Invariant:**

- at most one return capability per call stack level

\[ \text{cross-component call only allowed at EntryPoint} \]

\[ \text{loads and stores to the same component always allowed} \]

\[ \text{increment} \]

\[ \text{@(n+1)} \]

\[ \text{@(n+1)} \]

\[ \text{@(n+1)} \]
Secure compartmentalization property

∀ compromise scenarios.

∀ low-level attack from compromised $C_2 \downarrow$, $C_4 \downarrow$, $C_5 \downarrow$

∃ high-level attack from some fully defined $A_2$, $A_4$, $A_5$

follows from “structured full abstraction for unsafe languages” + “separate compilation”

[Beyond full abstraction, Juglaret, Hritcu, et al, draft’16]
Protecting higher-level abstractions

• ML abstractions we want to enforce with micro-policies
  – types, value immutability, opaqueness of closures, parametricity (dynamic sealing), GC vs malloc/free, ...

• F*: enforcing full specifications using micro-policies
  – some can be turned into contracts, checked dynamically
  – fully abstract compilation of F* to ML trivial for ML interfaces (because F* allows and tracks effects, as opposed to Coq)

• Limits of purely-dynamic enforcement
  – functional purity, termination, relational reasoning
  – push these limits further and combine with static analysis
Composing compilers and higher-level micro-policies

To compose compilers need
1. higher-level micro-policies
2. composing micro-policies

F* component
SecF*
SecML
CompSec
compiled F* component

ML μP
SecML μPolicy

C μP
SecML μPolicy

ASM (RISC-V+μP)
CompSec μPolicy
User-specified higher-level policies

• By composing more micro-policies we can allow **user-specified micro-policies for ML and C**
• Good news: **micro-policy composition is easy** since tags can be tuples
• But how do we ensure programmers won’t break security?
• Bad news: **secure micro-policy composition is hard!**
Secure micro-policy composition

• securely composing reference monitors is easy
  – ... as long as they can only stop execution
• micro-policies have richer interaction model:
  – monitor services: malloc, free, classify, declassify, ...
  – recoverable errors are similar
• composing micro-policies can break them
  – e.g. composing anything with IFC can leak
  – memory safety + compartmentalization
Secure compilation

• Solving conceptual challenges
  – Secure micro-policy composition
  – Higher-level micro-policies (for C and ML)
  – Formalizing security properties (i.e. attacker models)

• Building the first **efficient secure compilers** for realistic programming languages
  – C (CompCert): memory safety & compartmentalization
  – ML and F*: protecting higher-level abstractions

• Measuring & lowering the cost of secure compilation

• Showing that these compilers are indeed secure
  – Better verification and testing techniques
• Redesigned ML verification system [POPL’16]
  1. functional programming language with effects (like OCaml, F#, Standard ML, Haskell)
  2. deductive verification system based on SMT solvers (like FramaC, Why3, Dafny, Boogie, VCC, ESC/Java2)
  3. interactive proof assistant based on dependent types (like Coq, Lean, Agda)

• Working on language design, formal foundations, logical aspects, proof assistant, self-certification

• Main practical application:
  – verified reference implementation of upcoming TLS 1.3
Dependable property-based testing

• QuickCheck effective at finding bugs
• reducing the testing effort
  – language for property-based generators
• obtaining stronger confidence
  – polarized mutation testing
• providing stronger formal foundations
  – verified testing, generator synthesis(?)
• integrating testing in proof assistants
  – reducing the cost of interactive verification
Conclusion

• There is a pressing practical need for ...
  – more secure languages providing strong abstractions
  – more secure compiler chains protecting these abstractions
  – more secure hardware making the cost of all this acceptable
  – clear attacker models & strong formal security guarantees

• Building the first efficient secure compilers for realistic programming languages (C, ML, F*)

• Targeting micro-policies = new mechanism for hardware-accelerated tag-based monitors

Thank you!