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**

---

The glibc project thanks the Google Security Team and Red Hat for reporting the security impact of this issue, and Robert Holiday of Ciena for reporting the related bug 18665.
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
高效安全编译到微策略

2nd 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

1st 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 Sidiroglou-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**
- words in memory and registers are all tagged

*Conceptual model, our hardware implements this efficiently*
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>

<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>

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

decode(mem[1]) = store r0 r1

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

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

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

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

\[ !p \leftarrow 7 \]

free \( p \)

\[ 0@M(c, i) \quad 0@M(c, i) \quad \ldots \quad 0@M(c, i) \quad 7@M(c', i) \]

\[ q \leftarrow p + k \]

\[ q \leftarrow p + k \]

\( c \neq c' \)

\[ c \neq c' \] out of bounds

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

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

\[ \text{color of region} \]

\[ \text{tag of content} \]
Memory safety micro-policy

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

\[ q \leftarrow p + k \]

\[ c \neq c' \]

\[ q < 42 \]

\[ \text{out of bounds} \]

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

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

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

Intel MPX cannot detect this
Efficiently executing micro-policies

lookup $\downarrow$ zero overhead hits!

hardware cache
Efficiently executing micro-policies

lookup → 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)

More details
[ASPLOS’15]
Formal verification in Coq

Memory safe abstract machine

correctly implements

Symbolic machine

Micro-policy

correctly implements

Concrete machine

Rule cache

Monitor

memory safety micro-policy

correctly implements*

memory safety monitor

*only proved for IFC (verified DSL compiler)

Generic Framework

ASM

[POPL’14, Oakland’15]
Is this secure?

**Concrete machine**

- Correctly implements

**Symbolic machine**

- Correctly implements

**Abstract machine for P**

- Secure

**P in \{IFC, CFI\}**

(e.g. noninterference)

* 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* component

SecF*

ML component

SecML

C variants

CompSec*

CompSec*

CompSec+

Compiled F* component

Compiled ML component

Compiled safe C component

Compiled legacy C component

Compiled legacy C comp

Compiled ASM (RISC-V+μP)

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

invariant: at most one return capability per call stack level

cross-component call only allowed at EntryPoint

loads and stores to the same component always allowed
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
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!