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
1. Secure semantics for low-level languages

2. Secure interoperability with lower-level code
   • Formally: fully abstract compilation
     – holy grail, enforcing abstractions all the way down
     – currently this would be way too expensive

• Key enabling technology: micro-policies
  – hardware-accelerated tag-based monitoring

Efficient Secure Compilation to Micro-Policies

2nd part of this talk (more speculative)

1st part of this talk
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
- 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>

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

`tpc` `tr0` `tr1` `tr2` `tm1`
Tag-based instruction-level monitoring

\[
\begin{array}{|c|c|}
\hline
\text{pc} & \text{tpc} \\
\hline
\text{r0} & \text{tr0} \\
\hline
\text{r1} & \text{tr1} \\
\hline
\text{r2} & \text{tr2} \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{mem}[0] & \text{tm0} \\
\hline
\text{mem}[1] & \text{tm1} \\
\hline
\text{mem}[2] & \text{tm2} \\
\hline
\text{mem}[3] & \text{tm3} \\
\hline
\end{array}
\]

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

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

\text{store}

\text{monitor}

\text{bad action stopped!}

\text{disallow}
Efficiently executing micro-policies

lookup  zero overhead hits!

hardware cache
Efficiently executing micro-policies

lookup misses trap to software produced “rule” cached

hardware cache
Micro-policies are cool!

- low level + fine grained: unbounded per-word metadata, checked & propagated on each instruction
- efficient: accelerated using hardware caching
- expressive: can enforce large number of policies
- flexible: tags and monitor defined by software
- 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. \( ++c \))

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

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

\[ q \leftarrow p + k \]

\[ q \leftarrow p + k \]

out of bounds

\[ !q \leftarrow 42 \]

\[ !q \leftarrow 42 \]

free \( p \)

\[ !p \leftarrow 7 \]

\[ \text{free } p \]

\[ \text{color of region} \]

\[ \text{tag of content} \]

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

\[ T_m ::= M(c, T_v) \mid F \quad \text{tags on memory} \]
Memory safety micro-policy

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

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

\[ q \leftarrow p + k \]

\[ c \neq c' \]

\[ !q < 42 \]

Free \( p \)

Use after free \( x \)

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

Micro-Policies + can adapt to new threats + expressive enough for efficient secure compilation

Intel MPX cannot detect this

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

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

tags on values
tags on memory
SECURE COMPILATION
Joint work with Yannis Jeglaret
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
Fully abstract compilation, intuition

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*

SecML

ML component

编译F* component

编译ML component

编译安全C组件

编译后C组件

编译后Legacy C组件

CompSec

CompSec+

CompSec+

CompSec+

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**
  - stack level linear return capability
- **...(n+1)**
  - changed color
- **...@EntryPoint**
  - cross-component call only allowed at EntryPoint
- **Store ra → *rm**
- **...(n+1)**
- **Load *rm → ra**
- **Jump ra**

Registers:
- **pc**
- **...**
- **ra**
- **rm**

Memory:
- **C1**
- **C2**

**Invariant:**
- at most one return capability per call stack level

**Loads and stores to the same component always allowed**

**Cross-component return only allowed via return capability**
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
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
  – better hardware, hybrid enforcement (static + dynamic), weaker properties (robust compilation)

• Showing formally that these compilers are indeed secure
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!