Efficient Formally Secure Compilers to a Tagged Architecture

ătălin Hrițcu
• devastating low-level vulnerabilities
• programming languages, compilers, and hardware architectures
• the world has changed
• 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
  – “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
  - "Shell" <carlos at redhat dot com>
    - Date: Tue, 16 Feb 2016
    - Subject: [PATCH] CVE-2015-7547 -- glibc getaddrinfo(): stack-based buffer overflow in...
• memory safe
• useful abstractions
  – not immune to low-level attacks
    – unsafe interoperability with low-level code
      • enforcing abstractions all the way down too expensive
1. Secure semantics for low-level languages
2. Secure interoperability with lower-level code
   • fully abstract compilation
     – 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
• **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)
- **large tag**

<table>
<thead>
<tr>
<th>word</th>
<th>tag</th>
<th>tag[0]</th>
<th>tag[1]</th>
<th>tag[2]</th>
</tr>
</thead>
</table>

- **unbounded metadata**

<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>

<table>
<thead>
<tr>
<th>mem[0]</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem[1]</td>
<td>tag</td>
</tr>
<tr>
<td>mem[2]</td>
<td>tag</td>
</tr>
<tr>
<td>mem[3]</td>
<td>tag</td>
</tr>
</tbody>
</table>
Tag-based instruction-level monitoring

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

monitor

allow
<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 |

```plaintext
decode(mem[1]) = store r0 r1
```

**monitor** disallow bad action stopped!

```
store
```
Micro-policies are cool!

- low level + fine grained
- expressive
- flexible
- efficient
- secure
- real
• information flow control (IFC)
• monitor self-protection
• compartmentalization
• dynamic sealing
• heap memory safety
• code-data separation
• control-flow integrity (CFI)
• taint tracking

Expressiveness

Verified (in Coq)
Evaluated (<10% runtime overhead)

[Oakland’13, POPL’14]
[Oakland’15]
[ASPLOS’15]
• **Heap memory safety**
  – spatial violations
  – temporal violations
  – heap-allocated data

• **unforgeable capabilities**
Memory safety micro-policy

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

\[ \text{free } p \]

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

Intel MPX cannot detect this

free p

use after free

out of bounds
Efficiently executing micro-policies

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

Found zero overhead hits!
Efficiently executing micro-policies

produced “rule” cached
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)

50% -> 7%

220% -> 60% (40% now)

% Energy Ovhd

% Runtime Ovhd

% Area Ovhd

[ASPLOS'15]

spending silicon
Formal verification in Coq

Memory safe abstract machine

Symbolic machine

Micro-policy

Correctly implements

Correctly implements

Concrete machine

Rule cache

Monitor

Correctly implements

Memory safety micro-policy

Memory safety monitor

Generic Framework

[POPL’14, Oakland’15]

*Only proved for IFC (verified DSL compiler)
Abstract machine for $P$ correctly implements Symbolic machine correctly implements Concrete machine Rule cache Monitor monitor for $P$ 

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

extrinsic definition of memory safety

[Alpha is for address, Azevedo de Amorim et al, draft 2015]
SECURE COMPILATION
Joint work with Yannis Juglaret
• **Goal:** 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
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*
compiled F*
ML component
SecML
CompSec
+
compiled ML component
SecML
CompSec
+
safe C component
compiled safe C component
CompSec
+
compiled legacy C component
ASM component
ASM
C variants
(RISC-V+μP)
memory safe

μ
• Break up software into mutually distrustful components running with minimal privileges and interacting only via well-defined interfaces.

• Limit the damage just to 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 a better interaction model.

• We also aim to show security formally.
• add components with typed interfaces to C
  
  – each component’s memory tagged with unique color
  – procedure entry points tagged with procedure’s type

• component isolation
  – procedure call discipline
  – stack discipline for returns
  – type safety

Compartmentalization micro-policy

Jump \( \rightarrow \) \( \star \) \( \rightarrow \)

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

\[ \text{cross-component return only allowed via return capability} \]

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

\[ \text{invariant: at most one return capability per call stack level} \]
∀ low-level attack from compromised $C_2$, $C_4$, $C_5$

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

$\sim$ does not imply $L \sim H$

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

[Beyond full abstraction, Juglaret, Hritcu, et al, draft’16]
• ML abstractions we want to enforce with micro-policies

• F*: enforcing full specifications using micro-policies
  contracts,
  trivial for ML interfaces

• Limits of purely-dynamic enforcement
  — push these limits further and combine with static analysis
To compose compilers need
1. higher-level micro-policies
2. composing micro-policies
• **user-specified micro-policies for ML and C**

• **micro-policy composition is easy**

• programmers won’t

• **secure micro-policy composition is hard!**

\[
\begin{align*}
+\mu P & \quad \mu \quad \mu \\
+\mu P & \quad \mu \quad \mu \quad \mu \quad \mu \\
\mu & \quad \mu \quad \mu \quad \mu
\end{align*}
\]
Secure

- securely composing reference monitors is easy
  - can only stop execution
- richer interaction
  - monitor services:
  - recoverable errors

- composing micro-policies can break them
Secure compilation

• Solving conceptual challenges
  – Secure micro-policy composition
  – Higher-level micro-policies
  – Formalizing security properties

• 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
2. deductive verification system based on SMT solvers
3. interactive proof assistant based on dependent types

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

- verified reference implementation of upcoming TLS 1.3
• Dependable property-based testing
  • QuickCheck effective at finding bugs
  • reducing the testing effort
  • obtaining stronger confidence
  • providing stronger formal foundations
  • integrating testing in proof assistants
• There is a pressing practical need for ...
  – more secure languages
  – more secure compiler chains
  – more secure hardware
  – clear attacker models strong formal security guarantees
• efficient secure compilers realistic programming languages
• Targeting micro-policies

Thank you!