Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu

Inria Paris
• devastating low-level vulnerabilities
• programming languages, compilers, and hardware architectures
  – \texttt{m m q b \Phi q v q q ttPhi}
  – \texttt{m q btPhi x q b \max}
• the world has changed 1905 0751
  – \texttt{btPhi x q q v q q ttPhi ttPhi m m}
  – q q
• 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
  – 1904; m α q b m m m m ( m m v q F t t i q m m F )
  – 1905; q b b m b t t i q α ( “Spending silicon to improve security”)
Unsafe low-level languages

• B 0751 ( m B undefined behavior
  – mbtttnfft ttt q q v b b m
  – b q ffffq x ttt nfft
    ttn m qv x m m

• Programmers bear the burden for security
  – yttt v q bttw b ...:

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

DNS queries

vulnerable since May 2008

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  \( b \) (Java, OCaml, Haskell)

• useful abstractions  \( q v q mfft bttq b \)
  – F B x q b m tti tti x :::

• not immune to low-level attacks
  – \( qfft qtt\)n x mB q b m\(\)x
  – 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
   - q x; fully abstract compilation
     - x ffq m φ nfft q b m v x v m
     - currently this would be way too expensive

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

Efficient Secure Compilation to Micro-Policies

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

1\textsuperscript{st} 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
  - words in memory and registers are all tagged

- \[ \text{conceptual model, our hardware implements this efficiently} \]

- \[ \text{unbounded metadata} \]

- \[ \text{not addressable} \]

- \[ \text{metadata (protected meta-data space)} \]

- \[ \text{bq b b m v q} \]
Tag-based instruction-level monitoring

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

monitor

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

\text{pc} \leftarrow \text{r0}

\text{bad action stopped!}
Efficiently executing micro-policies...

Hardware cache lookup found zero overhead hits!
Efficiently executing micro-policies

tpc t1 t2 t3 tci

produced “rule” cached

q v q
- **low level + fine grained**: 

  - low level: unbounded per word metadata, checked & propagated on each instruction
  - fine grained: efficient, accelerated using hardware

- **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
• 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, POPL’14]

q \quad m

• m q m v b m q B (Oakland’13, POPL’14]
• m q q b m
• b q m m
• xm b nfft
• q x x
• b q m
• b mq v m fftq x B (Oakland’15]
• m q b nfft
• :::

ttt

09 qtttm q (ASPLOS’15]
• **Heap memory safety**
  – **spatial violations**; q mfft v q mfft t t t t m
  – **temporal violations**; t t t q q m q
  – **qheap-allocated data**

• **m q b unforgeable capabilities**
  – b m m x m m m q q f f t m
    • x b m f f t q f f t m q
    • x b x m f f t m f f t m m q q f f t m
Memory safety micro-
policy

```c
p ← malloc k

q, b

:fft b()

!p ← 7

free p
```

```
0

T v ::= i | ptr(c)       tags on values

T m ::= M(c, T v) | F     tags on memory

✓  ✓  ✓  ✓  ✓

\[
\text{color of region}
\]

\[
\text{tag of content}
\]

out of bounds

✓  ✓  ✓  ✓  ✓

✓  ✓  ✓  ✓  ✓
\( q \leftarrow p + k \)

\( !q \leftarrow 42 \)

free \( p \)
SECURE COMPILATION
Joint work with Yannis Juglaret
Secure compilation

• Goal: first efficient secure compilers for realistic programming languages

1. Secure semantics for low-level languages
   – $B \psi \quad x \quad m \quad b \quad q \quad m \quad m$

2. Secure interoperability with lower-level code
   – $B \quad m \{ \quad q \quad b \quad m \quad x \quad q \quad ( \quad q \quad q \quad t \quad t \quad q \quad m \quad q \quad m$

• $q \quad x$; fully abstract compilation
   – $m \quad \phi \quad m\text{fft} \quad q \quad b \quad m \quad v \quad x \quad v \quad m$
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 safe C component
CompSec

+ compiled legacy C component
ASM

C variants
ASM
(RISC-V+μP)

memory safe
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
• **V m add components with typed interfaces to C**

• **B q :fft B q ( m q q q q fft m q b m q m v q x fft**
  – each component’s memory tagged with unique color
  – procedure entry points tagged with procedure’s type

• **bq bx m φ nfft**
  – component isolation
  – procedure call discipline  m φ m (  
  – stack discipline for returns  m q q ttmb  
  – type safety  mbq b m m m m q b m

Compartmentalization micro-policy

Jal

Load \rightarrow \text{EntryPoint}

Jump \rightarrow \text{pc}

\text{pc} \rightarrow \text{ra} \rightarrow \text{rm} \rightarrow \text{ra}

\text{pc} \rightarrow \text{rm} \rightarrow \text{ra} \rightarrow \text{@n} \rightarrow \text{@}(n+1)

\text{Store} \rightarrow \text{rm} \rightarrow \text{ra} \rightarrow \text{@n} \rightarrow \text{@}(n+1)

cross-component call only allowed at EntryPoint

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

invariant:

m

m

q fft

q

m

q

\text{ttmb}

\text{qb}

\text{b}

\text{bx}
∀b q b m q :

∀v ∀fft B_0 B_1 B_2 B_3 B_4 B_0 B_1 B_2 B_3 B_4

∃ttt x m 1 3 4

∀ compromise scenarios.

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
  \[ x \quad \text{ttt} \quad \text{ttt} \quad x \quad \text{ttt} \quad \text{m} \quad b \quad \text{ttt} \quad q \quad q \quad b \quad x \quad x \quad \text{m} \quad b \quad \text{mfft} \quad \text{FB} \quad b \quad q \quad \text{:::} \]

• F*: enforcing full specifications using micro-policies
  \[ b \quad m \quad \text{ttt} \quad m \quad \text{contracts, b b x m b x} \]

  \[ \text{ttt} \quad x \quad q \quad b \quad b \quad m \quad \{ \quad \text{trivial for ML interfaces} \quad b \quad \text{ttt} \quad \{ \quad v \quad m \quad q \quad b \quad b \quad \text{B (} \]

• Limits of purely-dynamic enforcement
  \[ \text{ttt} \quad b \quad m \quad \text{ttt} \quad x \quad q \quad m \quad m \quad q \quad m \quad q \quad \text{mmfft} \]

  – push these limits further and combine with static analysis
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
  – Better hardware, hybrid enforcement (static + dynamic), weaker properties (robust compilation)

• Showing formally that these compilers are indeed secure
• 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
• Targeting micro-policies

Thank you!