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
Today’s processors are mindless bureaucrats
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**

- Date: Tue, 16 Feb 2016 15:09:25 -0400
- Subject: [PATCH] CVE-2015-7547 --- glibc getaddrinfo stack-based buffer overflow
- Authentication-results: sourceware.org; auth=none
- References: <56C32C20 dot 1070006 at redhat dot com>

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

5
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

1st part of this talk

2nd part of this talk (more speculative)

Secure Compilation
Formal methods & architecture systems

Current team:
– Inria Cătălin Hrițcu
– UPenn
– Portland State
– MIT:
– Industry

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

- Add large tag to each machine word
- Words in memory and registers are all tagged

*Conceptual model, our hardware implements this efficiently*

Unbounded metadata
(protected meta-data space)
(not addressable)
Tag-based instruction-level monitoring

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

monitor allow tpc tr0 tr1 tr2 tm1
decode(mem[1]) = store r0 r1

bad action stopped!

store
Efficiently executing micro-policies

```
<table>
<thead>
<tr>
<th>tpc</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>tci</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Efficiently executing micro-policies

lookup misses trap to software
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
• 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'15] [Oakland'13, POPL'14] [ASPLOS'15]
Flexibility (by example)

• Heap memory safety
  – 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
p ← malloc

!p ← 7

q ← p + 1

!q ← 42
Oracle Silicon Secured Memory (2016) similar, but with only 16 colors

Intel MPX cannot detect this

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

• 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*
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

(RISC-V+μP)

memory safe

C variants
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
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

\[ \text{EntryPoint} \]

- Jump
- \( \text{pc} \rightarrow \star \)
- \( \text{ra} \rightarrow \star \)

\[ \text{Load} \]

\[ \text{Store} \]

Cross-component call only allowed at \( \text{EntryPoint} \)

\[ \text{linear return capability} \]

\[ \text{stack level} \]
- \( \text{current color} \)
- \( \text{changed color} \)
- \( \text{increment loads and stores to the same component always allowed} \)

\[ \text{invariant: at most one return capability per call stack level} \]

\[ \text{cross-component return only allowed via return capability} \]
∀ low-level attack from compromised $C_2$, $C_4$, $C_5$;
∃ high-level attack from some fully defined $A_2$, $A_4$, $A_5$.

$\sim \not\equiv L \not\equiv H \↯ \↯ \↯ \↯ \↯$

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
• 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!