Formally Secure Compilation of Unsafe Low-level Components

Cătălin Hrițcu
Inria Paris

https://secure-compilation.github.io
Parcurs profesional

• 2001 - 2005 - Infoiași - student la licență
• 2005 - 2011 - Saarland University - MSc & PhD
• 2011 - 2013 - U. of Pennsylvania - PostDoc with Benjamin Pierce, DARPA CRASH/SAFE
• 2013 - acum - Inria Paris - Cercetător
• 2017 - 2021 - ERC Starting Grant SECOMP - PI
• 2017 - 2020 - DARPA SSITH/HOPE - coPI
Computers are insecure

- devastating low-level vulnerabilities
- teasing out 2 important security problems:
  1. inherently insecure low-level languages
     - memory unsafe: any buffer overflow can be catastrophic allowing remote attackers to gain complete control
  2. unsafe interaction with unsafe code
     - even code written in safer languages has to interoperate with unsafe code
     - unsafe interaction: safety guarantees lost
How did we get here?

• 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 (2017 vs 1972*)
  – security matters, hardware resources abundant
  – time to revisit some tradeoffs

* “...the number of UNIX installations has grown to 10, with more expected...”
  -- Dennis Ritchie and Ken Thompson, June 1972
Key enabler: Micro-Policies

software-defined, hardware-accelerated, tag-based 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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mem[0]</th>
<th>tm0</th>
</tr>
</thead>
<tbody>
<tr>
<td>“store r0 r1”</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>

store

monitor

allow

disallow

software monitor’s decision is hardware cached

policy violation stopped!
(e.g. out of bounds write)
Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
- **flexible**: tags and monitor defined by software
- **efficient**: software decisions hardware cached
- **expressive**: complex policies for secure compilation
- **secure** and **simple** enough to verify security in Coq
- **real**: FPGA implementation on top of RISC-V
Expressiveness

- information flow control (IFC) [POPL’14]
- monitor self-protection
- protected compartments
- 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]

Way beyond MPX, SGX, SSM, etc
Micro-Policies Project

- **Formal methods & architecture & systems**
- **Previous:** DARPA CRASH/SAFE (2011-2014)
- **Current:** DARPA SSITH/HOPE (2017-2020)
- **PIs:**
  - *Draper Labs:* Arun Thomas, Chris Casinghino
  - *Dover Microsystems:* Greg Sullivan
  - *DornerWorks:* Nathan Studer, David Johnson
  - *UPenn:* André DeHon, Benjamin Pierce
  - *Inria Paris:* Cătălin Hrițcu
  - *Portland State:* Andrew Tolmach
  - *MIT:* Howie Shrobe
ERC SECOMP Grand Challenge (2017-2021)

Use micro-policies to build the first efficient formally secure compilers for realistic programming languages

1. Provide secure semantics for low-level languages
   - C with protected components and memory safety

2. Enforce secure interoperability with unsafe code
   - ASM, C, and Low* [= safe C subset embedded in F* for verification]
Goal: achieving secure compilation at scale

**Low* language**
(safe C subset in F*)

**C language**
+ components
+ memory safety

**ASM language**
(RISC-V + micro-policies)

![Diagram](image)

Protecting component boundaries
Formally Secure Compilation of Unsafe Low-level Components
Compartmentalization for unsafe, low-level languages

• **Add components to C-like language**
  - interacting only via *strictly enforced* interfaces

• **Secure compilation chain**
  - use low-level security mechanisms to *efficiently enforce*:
    component separation, call and return discipline, ...

• **Interesting attacker model**
  - mutual distrust, dynamic compromise, least privilege
    • e.g. dynamic compromise = "each component should be protected from all the others until it becomes compromised and starts attacking the remaining uncompromised components"

**Goal:** Build this

**Goal:** Formalize this
Formally secure compilation

holly grail of preserving security all the way down

Benefit: sound security reasoning in the source language
forget about compilation chain (linker, loader, runtime)
forget that libraries are written in a lower-level language
Fully abstract compilation
preservation of observational equivalence
Undefined behavior

```c
#include <string.h>
int main (int argc, char **argv)
{
    char c[12];
    strcpy(c, argv[1]);
    return 0;
}
```

Buffer overflow

```
$ gcc target.c -fno-stack-protector
$ ./a.out haha
$ ./a.out hahahahahahahahahahahahahahaha
zsh: segmentation fault (core dumped)
```
Source reasoning vs undefined behavior

• **Source reasoning**
  = We want to reason formally about security with respect to source language semantics

• **Undefined behavior**
  = can't be expressed at all by source language semantics!

• **Observational equivalence doesn't work with undefined behavior!?**
  – int buf[5]; buf[42] ~ int buf[5]; buf[43]?

• Can we somehow avoid undefined behavior?
Full abstraction
with mutually distrustful components

∀ compromise scenarios.

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

if $C_1, C_3, D_1, D_3$ fully defined and

∃ low-level attack from compromised $C_2\downarrow, C_4\downarrow, C_5\downarrow$

Limitation: static compromise model: $C_1, C_3, D_1, D_3$ get guarantees only if perfectly safe
(i.e. fully defined = do not exhibit undefined behavior in any context)

This is the most we were able to achieve for full abstraction!

[Beyond Good and Evil - Juglaret, Hrițcu, et al, CSF’16]
Static compromise not good enough

component $C_0$ {
    export valid;
    valid(data) { ... }
}

component $C_1$ {
    import E.read, C_2.init, C_2.process;
    main() {
        C_2.init();
        x := E.read();
        y := C_1.parse(x);  // ($V_1$) can UNDEF if $x$ is malformed
        C_2.process(x, y);
    }
    parse(x) { ... }
}

component $C_2$ {
    import E.write, C_0.valid;
    export init, process;
    init() { ... }
    process(x, y) { ... }  // ($V_2$) can UNDEF if not initialized
New secure compilation criterion: Robust Compilation

∀ (bad, attack) trace \( t \)

- high-level attacker causing \( t \)
- low-level attacker causing \( t \)
- compiler

robust trace property preservation (robust = in adversarial context)

intuition:
- stronger than compiler correctness
- seems weaker than full abstraction + compiler correctness
- less extensional than full abstraction

Advantages: easier to realistically achieve and prove at scale
useful: preservation of invariants and other integrity properties
more intuitive to security people (generalizes to hyperproperties!)
extends to unsafe languages (supporting dynamic compromise)
∃ a **dynamic compromise scenario** explaining \( t \) in source language for instance \( \exists [A_1, A_2] \) leading to the following compromise sequence:

1. \( C_0 \) \( \downarrow \) \( C_1 \downarrow \) \( C_2 \downarrow \) \( t \)

\( \Downarrow \)

2. \( C_0 \) \( \downarrow \) \( A_1 \) \( C_2 \downarrow \) \( m_2; Undef(C_2) \)

\( \Downarrow \)

3. \( C_0 \) \( \downarrow \) \( A_1 \) \( A_2 \downarrow \) \( t \)

**Trace is very helpful**
- detect undefined behavior
- rewind execution

[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]
Now we know what these words mean!
(at least in the setting of compartmentalization for unsafe low-level languages)

Mutual distrust  
\[ C_1 \quad A_2 \quad C_3 \quad A_4 \quad A_5 \]

Dynamic compromise  
\[ C_0 \quad A_1 \quad C_2 \downarrow m_2; \text{Undef}(C_2) \]

Least privilege  
\[ i_1 \quad i_2 \quad i_3 \]

[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]
Simple Secure Compilation Chain

Verified (in Coq)

Compartmentalized unsafe source

Buffers, procedures, components interacting via strictly enforced interfaces

Compartmentalized abstract machine

Simple RISC abstract machine with build-in compartmentalization

Software fault isolation

Micro-policy machine

Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline

Standard machine

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline

Systematically tested (with QuickChick)
Beyond trace properties

(Trace) property = set of traces
Hyperproperty = set of sets of traces

Legend


Beyond trace properties

- Robust Hyperproperty Preservation
- Robust 2-Hypersafety Preservation
- Robust Hypersafety Preservation
- Robust K-Hypersafety Preservation
- Robust 2-Subset-Closed Hyperproperty Preservation
- Robust K-Subset-Closed Hyperproperty Preservation

Legend

(Trace) property = set of traces
Hyperproperty = set of sets of traces

Full abstraction lives around here (if we move VP after 3Cs)
i.e. preserving relational 2-hypersafety

Back-translating contexts

Back-translating finite trace prefixes

Legend

(Trace) property = set of traces
Hyperproperty = set of sets of traces

Full abstraction lives around here (if we move VP after 3Cs)
i.e. preserving relational 2-hypersafety
Compartmentalization mechanisms

• **practically deployed ones**
  – process-level privilege separation (all web browsers)
  – software fault isolation (SFI, Google Native Client)
  – hardware enclaves (Intel SGX, ARM TrustZone)

• **and more on drawing boards:**
  – WebAssembly (WASM)
  – capability machines (CHERI)
  – tagged architectures (micro-policies)