SECOMP
Efficient Formally Secure Compilers
to a Tagged Architecture

Cătălin Hrițcu
Inria Paris

(visiting researcher at Microsoft until end of November)
(member of Everest expedition)

https://secure-compiler.github.io/
SECOMP
Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu
Inria Paris

(visiting researcher at Microsoft until end of November)
(member of Everest expedition)

https://secure-compilation.github.io/
Computers are insecure

• devastating low-level vulnerabilities
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
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

* “...the number of UNIX installations has grown to 10, with more expected...”
  -- Dennis Ritchie and Ken Thompson, June 1972
Hardware architectures

• Today’s processors are mindless bureaucrats
  – “write past the end of this buffer” ... yes boss!
  – “jump to this untrusted integer” ... right boss!
  – “return into the middle of this instruction” ... sure boss!
Hardware architectures

• **Today’s processors are mindless bureaucrats**
  – “write past the end of this buffer” ... *yes boss!*
  – “jump to this untrusted integer” ... *right boss!*
  – “return into the middle of this instruction” ... *sure boss!*

• **Software bears most of the burden for security**
Hardware architectures

• Today’s processors are mindless bureaucrats
  – “write past the end of this buffer”       ... yes boss!
  – “jump to this untrusted integer”         ... right boss!
  – “return into the middle of this instruction” ... sure boss!

• 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)
Hardware architectures

• Today’s processors are mindless bureaucrats
  – “write past the end of this buffer” ... yes boss!
  – “jump to this untrusted integer” ... right boss!
  – “return into the middle of this instruction” ... sure boss!

• 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
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
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**

- From: "Carlos O'Donell" <carlos at redhat dot com>
- To: GNU C Library <libc-alpha at sourceware dot org>
- Date: Tue, 16 Feb 2016 09:09:52 -0500
- 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.
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** 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 (at a cost)
Safer high-level languages?

• **memory safe** (at a cost)

• **useful abstractions** for writing secure code:
  – GC, type abstraction, modules, immutability, ...
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
Teasing out 2 different problems

• **1. inherently insecure low-level languages**
  
  – **memory unsafe**: any buffer overflow can be catastrophic allowing remote attackers to gain complete control
Teasing out 2 different problems

• 1. inherently insecure low-level languages
  – memory unsafe: any buffer overflow can be catastrophic allowing remote attackers to gain complete control

• 2. unsafe interoperability with lower-level code
  – even code written in safer high-level languages has to interoperate with insecure low-level libraries
  – unsafe interoperability: all high-level safety guarantees lost
Key enabler: Micro-Policies
software-defined, hardware-accelerated, tag-based monitoring
Key enabler: Micro-Policies

software-defined, hardware-accelerated, tag-based monitoring

```

pc  mem[0]
r0  "store r0 r1"
r1  mem[2]
    mem[3]
```
Key enabler: Micro-Policies
software-defined, hardware-accelerated, tag-based monitoring
Key enabler: Micro-Policies
software-defined, hardware-accelerated, tag-based monitoring
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

tpc’ tm3’
Key enabler: Micro-Policies
software-defined, hardware-accelerated, tag-based monitoring

store

monitor

allow
Key enabler: Micro-Policies
software-defined, hardware-accelerated, tag-based monitoring

store

monitor

software monitor’s decision is hardware cached
**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>

```
mem[0] | tm0
---|---
"store r0 r1" | tm1
mem[2] | tm2
mem[3] | tm3
```

**Software-defined, hardware-accelerated, tag-based monitoring**

- **store**
- **monitor**
- **disallow**

**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
**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
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]
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
- ...

...
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
• ...

Way beyond MPX, SGX, SSM, etc
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
- ...

Way beyond MPX, SGX, SSM, etc

Verified (in Coq)
[Oakland’15]
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 [ASPLOS’15]

Verified (in Coq) [Oakland’15]

Evaluated (<10% runtime overhead)
Micro-Policies team

• Formal methods & architecture & systems
• Current team:
  – *Inria Paris*: Cătălin Hrițcu, Marco Stronati
    (until recently Yannis Juglaret, Boris Eng)
  – *UPenn*: André DeHon, Benjamin Pierce,
    Arthur Azevedo de Amorim, Nick Roessler
  – *Portland State*: Andrew Tolmach
  – *MIT*: Howie Shrobe,
    Stelios Sidiropoglou-Douskos
  – *Industry*: Draper Labs, Bluespec Inc
Micro-Policies team

• Formal methods & architecture & systems

• Current team:
  – Inria Paris: Cătălin Hrițcu, Marco Stronati (until recently Yannis Juglaret, Boris Eng)
  – UPenn: André DeHon, Benjamin Pierce, Arthur Azevedo de Amorim, Nick Roessler
  – Portland State: Andrew Tolmach
  – MIT: Howie Shrobe, Stelios Sidiropoglou-Douskos
  – Industry: Draper Labs, Bluespec Inc

• Spinoff of past project: DARPA CRASH/SAFE (2011-2014)
SECOMP grand challenge

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

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
SECOMP grand challenge

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 lower-level code
   – ASM, C, and F* [= OCaml/F# + verification]
Formally verify: **full abstraction**

holy grail of secure compilation, enforcing abstractions all the way down
Formally verify: **full abstraction**

holy grail of secure compilation, enforcing abstractions all the way down
Formally verify: **full abstraction**

holy grail of secure compilation, enforcing abstractions all the way down

- **program behavior**
  - **compiler correctness** (e.g. CompCert) not enough
  - program behavior

- **source component**
  - **target component**
  - low-level attacker

  e.g. arbitrary machine code
Formally verify: **full abstraction**

holy grail of secure compilation, enforcing abstractions all the way down

- **program behavior**
  - **compiler correctness** (e.g. CompCert)
  - Not enough

- **source component**
  - High-level attacker

- **target component**
  - Low-level attacker

- **e.g. arbitrary machine code**
Formally verify: full abstraction

holy grail of secure compilation, enforcing abstractions all the way down

program behavior

compiler correctness (e.g. CompCert)

not enough

program behavior

source component

high-level attacker

full abstraction

target component

protected

low-level attacker

no extra power

e.g. arbitrary machine code
Formally verify: full abstraction

holly grail of secure compilation, enforcing abstractions all the way down

Benefit: sound security reasoning in the source language
Forget about compiler chain (linker, loader, runtime system)
Forget that libraries are written in a lower-level language
Formally verify: **full abstraction**

holy grail of secure compilation, enforcing abstractions all the way down

**Benefit:** sound security reasoning in the source language
forget about compiler chain (linker, loader, runtime system)
forget that libraries are written in a lower-level language

not efficiently achievable today
Fully abstract compilation, definition

∃ low-level attacker.

1\textsuperscript{st} high-level component

compiler

1\textsuperscript{st} compiled component

low-level attacker
Fully abstract compilation, definition

∃ low-level attacker

1\(^{st}\) high-level component

compiler

1\(^{st}\) compiled component

low-level attacker

2\(^{nd}\) high-level component

compiler

2\(^{nd}\) compiled component

low-level attacker
Fully abstract compilation, definition

∃ high-level attacker

∃ low-level attacker

compiler

1st high-level component

high-level attacker

1st compiled component

low-level attacker

∀

compiler

2nd high-level component

high-level attacker

2nd compiled component

low-level attacker
Fully abstract compilation, definition

\[ \exists \text{ high-level attacker} \quad 1^{\text{st}} \text{ high-level component} \quad \text{high-level attacker} \quad 2^{\text{nd}} \text{ high-level component} \quad \text{high-level attacker} \]

\[ \exists \text{ low-level attacker} \quad 1^{\text{st}} \text{ compiled component} \quad \text{low-level attacker} \quad 2^{\text{nd}} \text{ compiled component} \quad \text{low-level attacker} \]
SECOMP: achieving full abstraction at scale

**F* language**
(OCaml/F# + verification)

**C language**
+ memory safety
+ components
SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

miTLS*

KremSec
SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

miTLS*
KremSec
memory safe C component
SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components
SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

miTLS*
KremSec
CompSec*
memory safe C component
SECOMP: achieving full abstraction at scale

**F* language**  
(OCaml/F# + verification)

**C language**  
+ memory safety  
+ components

**ASM language**  
(RISC-V + micro-policies)
SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

protecting component boundaries
SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

protecting component boundaries
SECOMP: achieving full abstraction at scale

**F* language**
(OCaml/F# + verification)

**C language**
+ memory safety
+ components

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

Diagram: miTLS* connected to KremSec, KremSec to CompSec+, CompSec+ to memory safe C component, memory safe C component to legacy C component, legacy C component to ASM component. The diagram highlights protecting component boundaries.
SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

protecting higher-level abstractions

protecting component boundaries

stronger connection to Everest expedition

miTLS*
KremSec
CompSec*
CompSec

memory safe C component
legacy C component

ASM component
Protecting component boundaries

• Add mutually distrustful components to C
  – interacting only via strictly enforced interfaces
Protecting component boundaries

• Add mutually distrustful components to C
  – interacting only via strictly enforced interfaces

• CompSec compiler chain (based on CompCert)
  – propagate interface information to produced binary
Protecting component boundaries

• Add mutually distrustful components to C
  – interacting only via strictly enforced interfaces

• CompSec compiler chain (based on CompCert)
  – propagate interface information to produced binary

• Micro-policy simultaneously enforcing
  – component separation
  – type-safe procedure call and return discipline
Protecting component boundaries

• Add mutually distrustful components to C
  – interacting only via strictly enforced interfaces

• CompSec compiler chain (based on CompCert)
  – propagate interface information to produced binary

• Micro-policy simultaneously enforcing
  – component separation
  – type-safe procedure call and return discipline

• Interesting attacker model
  – extending full abs. to mutual distrust + unsafe source
Protecting component boundaries

• Add mutually distrustful components to C
  – interacting only via strictly enforced interfaces

• CompSec compiler chain (based on CompCert)
  – propagate interface information to produced binary

• Micro-policy simultaneously enforcing
  – component separation
  – type-safe procedure call and return discipline

• Interesting attacker model
  – extending full abs. to mutual distrust + unsafe source

Recent work, joint with Yannis Juglaret et al
Compartmentalization micro-policy

Compartmentalization micro-policy

Compartmentalization micro-policy

Compartmentalization micro-policy

Compartmentalization micro-policy

Compartmentalization micro-policy

- **C₁**
  - Jal r
  - ...
  - ...
  - ...@EntryPoint
  - Store rₐ → *rₘ
  - ...
  - Load *rₘ → rₐ
  - Jump rₐ

- **C₂**

**Memory**

**Registers**

- linear return capability
- @Ret n
- @(n+1)

- pc
- rₐ
- rₘ

loads and stores to the same component always allowed
Compartmentalization micro-policy

memory

C1

Jal r
...
...
...@EntryPoint
Store r_a → *r_m
...
Load *r_m → r_a
Jump r_a

C2

linear return capability

C registers

@Ret n

pc

(n+1)

r_a

r_m

@Ret n

(n+1)
Compartmentalization micro-policy

- **Jal r**
- ...
- ...
- @EntryPoint
- Store \( r_a \rightarrow *r_m \)
- ...
- Load \( *r_m \rightarrow r_a \)
- Jump \( r_a \)

**C_1**

**C_2**

**memory**

**registers**

**linear return capability**

**invariant:** at most one return capability per call stack level
Compartmentalization micro-policy

**Invariant:**
At most one return capability per call stack level

<table>
<thead>
<tr>
<th>memory</th>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jal r</td>
<td>registers</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...@EntryPoint</td>
<td></td>
</tr>
<tr>
<td>Store ( r_a \to \star r_m )</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Load ( \star r_m \to r_a )</td>
<td></td>
</tr>
<tr>
<td>Jump ( r_a )</td>
<td></td>
</tr>
</tbody>
</table>

\( C_1 \quad C_2 \)

linear return capability

\( \text{pc} \quad r_a \quad r_m \)

@Ret n

@Ret n

@\((n+1)\)
Compartmentalization micro-policy

memory

C1

Jal r
...
...
...@EntryPoint
Store ra \rightarrow *rm
...
Load *rm \rightarrow ra
Jump ra

C2

registers

\text{linear return capability}

\text{@Ret n}

\text{@Ret n}

\text{pc}

\text{ra}

\text{rm}

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

\text{invariant: at most one return capability per call stack level}
Secure compartmentalizing compilation (SCC)

∀ compromise scenarios.
Secure compartmentalizing compilation (SCC)

∀compromise scenarios.
Secure compartmentalizing compilation (SCC)

∀ compromise scenarios.

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

∃ high-level attack from some fully defined $A_2, A_4, A_5$
Secure compartmentalizing compilation (SCC)

∀ 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 Good and Evil, Juglaret, Hritcu, et al, CSF’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, ...
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)
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
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
SECOMP focused on dynamic enforcement but combining with static analysis can ...

• improve efficiency
  – removing spurious checks
  – e.g. turn off pointer checking for a statically memory safe component that never sends or receives pointers
SECOMP focused on dynamic enforcement but combining with static analysis can ...

- **improve efficiency**
  - removing spurious checks
    - e.g. turn off pointer checking for a statically memory safe component that never sends or receives pointers

- **improve transparency**
  - allowing more safe behaviors
    - e.g. statically detect which copy of linear return capability the code will use to return
  - in this case unsound static analysis is fine
SECOMP in a nutshell

- We need more secure languages, compilers, hardware
SECOMP in a nutshell

• We need more secure languages, compilers, hardware
• Key enabler: micro-policies (software-hardware protection)
• Grand challenge: the first efficient formally secure compilers for realistic programming languages (C and F*)
SECOMP in a nutshell

• We need more secure languages, compilers, hardware
• Key enabler: micro-policies (software-hardware protection)
• Grand challenge: the first efficient formally secure compilers for realistic programming languages (C and F*)
• Answering challenging fundamental questions
  – attacker models, proof techniques
  – secure composition, micro-policies for C
SECOMP in a nutshell

• We need more **secure languages, compilers, hardware**

• **Key enabler: micro-policies** (software-hardware protection)

• **Grand challenge:** the first efficient formally secure compilers for realistic programming languages (C and F*)

• **Answering challenging fundamental questions**
  – attacker models, proof techniques
  – secure composition, micro-policies for C

• **Achieving strong security properties like full abstraction**
  + testing and proving formally that this is the case
SECOMP in a nutshell

• We need more secure languages, compilers, hardware
• Key enabler: micro-policies (software-hardware protection)
• Grand challenge: the first efficient formally secure compilers for realistic programming languages (C and F*)
• Answering challenging fundamental questions
  – attacker models, proof techniques
  – secure composition, micro-policies for C
• Achieving strong security properties like full abstraction
  + testing and proving formally that this is the case
• Measuring & lowering the cost of secure compilation
SECOMP in a nutshell

• We need more **secure languages, compilers, hardware**

• **Key enabler:** micro-policies (software-hardware protection)

• **Grand challenge:** the **first efficient formally secure compilers** for realistic programming languages (C and F*)

• **Answering challenging fundamental questions**
  – attacker models, proof techniques
  – secure composition, micro-policies for C

• **Achieving strong security properties like full abstraction**
  + testing and proving formally that this is the case

• **Measuring & lowering the cost of secure compilation**

• Most of this is **vaporware** at this point but ...
  – building a community, looking for collaborators, and hiring
Collaborators & Community

• Traditional collaborators from Micro-Policies project
  – UPenn, MIT, Portland State, Draper Labs
Collaborators & Community

• Traditional collaborators from Micro-Policies project
  – UPenn, MIT, Portland State, Draper Labs

• Several other researchers working on secure compilation
  – Deepak Garg (MPI-SWS), Frank Piessens (KU Leuven),
    Amal Ahmed (Northeastern), Cedric Fournet & Nik Swamy (MSR)
Collaborators & Community

- Traditional collaborators from Micro-Policies project
  - UPenn, MIT, Portland State, Draper Labs
- Several other researchers working on secure compilation
  - Deepak Garg (MPI-SWS), Frank Piessens (KU Leuven), Amal Ahmed (Northeastern), Cedric Fournet & Nik Swamy (MSR)
- Secure compilation meetings (very informal)
  - 1st at Inria Paris in August 2016
  - 2nd in Paris on 15 January 2017 before POPL at UPMC
  - Work in progress proposal for Dagstuhl seminar in 2018
  - build larger research community, identify open problems, bring together communities (hardware, systems, security, languages, verification, ...)

23
BACKUP SLIDES
• Looking for excellent **interns, PhD students, PostDocs, starting researchers**, and **engineers**

• We can also support outstanding candidates in the **Inria permanent researcher competition**
Beyond full abstraction

- Is full abstraction always the right notion of secure compilation? The right attacker model?
Beyond full abstraction

• Is full abstraction always the right notion of secure compilation? The right attacker model?

• **Similar properties**
  – secure compartmentalizing compilation (SCC)
  – preservation of hyper-safety properties [Garg et al.]
Beyond full abstraction

• Is full abstraction always the right notion of secure compilation? The right attacker model?

• **Similar properties**
  – secure compartmentalizing compilation (SCC)
  – preservation of hyper-safety properties [Garg et al.]

• **Strictly weaker properties** (easier to enforce!):
  – robust compilation (integrity but no confidentiality)
Beyond full abstraction

• Is full abstraction always the right notion of secure compilation? The right attacker model?

• **Similar properties**
  – secure compartmentalizing compilation (SCC)
  – preservation of hyper-safety properties [Garg et al.]

• **Strictly weaker properties** (easier to enforce!):
  – robust compilation (integrity but no confidentiality)

• **Orthogonal properties**:
  – memory safety (enforcing CompCert memory model)
What secure compilation adds over compositional compiler correctness

- **mapping back arbitrary low-level contexts**
- **preserving integrity properties**
  - robust compilation phrased in terms of this
- **preserving confidentiality properties**
  - full abstraction and preservation of hyper-safety phrased in terms of this
- **stronger notion of components and interfaces**
  - secure compartmentalizing compilation adds this
Verification and testing

• So far all secure compilation work on paper
  – but one can’t verify an interesting compiler on paper
Verification and testing

• So far all secure compilation work on paper
  – but one can’t verify an interesting compiler on paper
• SECOMP will use proof assistants: Coq and F*
Verification and testing

• So far all secure compilation work on paper
  – but one can’t verify an interesting compiler on paper
• SECOMP will use proof assistants: Coq and F*
• Reduce effort
  – better automation (e.g. based on SMT like in F*)
  – integrate testing and proving (QuickChick and Luck)
Verification and testing

- So far all secure compilation work **on paper**
  - but one can’t verify an interesting compiler on paper
- SECOMP will use **proof assistants**: Coq and F*
- **Reduce effort**
  - better automation (e.g. based on SMT like in F*)
  - integrate testing and proving (QuickChick and Luck)
- **Problems not just with effort-scale**
  - devising good **proof techniques** for full abstraction
    is a hot research topic of its own
Micro-policies:
remaining fundamental challenges
Micro-policies:
remaining fundamental challenges

• Micro-policies for C
  – needed for vertical compiler composition
  – will put micro-policies in the hands of programmers
Micro-policies: remaining fundamental challenges

• Micro-policies for C
  – needed for vertical compiler composition
  – will put micro-policies in the hands of programmers

• Secure micro-policy composition
  – micro-policies are interferent reference monitors
  – one micro-policy’s behavior can break another’s guarantees
    • e.g. composing anything with IFC can leak
Beyond full abstraction

• Is full abstraction always the right notion of secure compilation? The right attacker model?
Beyond full abstraction

• Is full abstraction always the right notion of secure compilation? The right attacker model?

• **Similar properties**
  – secure compartmentalizing compilation (SCC)
  – preservation of hyper-safety properties [Garg et al.]
Beyond full abstraction

• Is full abstraction always the right notion of secure compilation? The right attacker model?

• **Similar properties**
  – secure compartmentalizing compilation (SCC)
  – preservation of hyper-safety properties [Garg et al.]

• **Strictly weaker properties** (easier to enforce!):
  – robust compilation (integrity but no confidentiality)
Beyond full abstraction

• Is full abstraction always the right notion of secure compilation? The right attacker model?

• **Similar properties**
  – secure compartmentalizing compilation (SCC)
  – preservation of hyper-safety properties [Garg et al.]

• **Strictly weaker properties** (easier to enforce!):
  – robust compilation (integrity but no confidentiality)

• **Orthogonal properties**: 
  – memory safety (enforcing CompCert memory model)
Composing compilers and higher-level micro-policies

\[
\text{F*} \xrightarrow{\text{SecKremlinF*}} \text{C+μP} \xrightarrow{\text{CompSec}^+} \text{ASM (RISC-V+μP)} \xrightarrow{} \text{compiled F* component}
\]
Composing compilers and higher-level micro-policies

F* component
SecKremlinF*

F*

F* component
CompSec+
compiled F* component

SecF* μPolicy

CompSec* μPolicy

C+μP

CompSec+

ASM (RISC-V+μP)
Composing compilers and higher-level micro-policies

To compose compilers need
1. higher-level micro-policies
2. composing micro-policies
User-specified higher-level policies

• By composing more micro-policies we can allow user-specified micro-policies for C
User-specified higher-level policies

- By composing more micro-policies we can allow **user-specified micro-policies for C**
- **Good news:**
  - **micro-policy composition is easy** since tags can be tuples

```
C+μP
  SeKremlin μPolicy ➔ user-specified C μPolicy
ASM (RISC-V+μP) ➔ CompSec μPolicy ➔ user-specified ASM μPolicy
```
User-specified higher-level policies

• By composing more micro-policies we can allow user-specified micro-policies for C
• Good news: micro-policy composition is easy since tags can be tuples
• But how do we ensure programmers won’t break security?
User-specified higher-level policies

• By composing more micro-policies we can allow **user-specified micro-policies for C**
• Good news: **micro-policy composition is easy** since tags can be tuples
• But how do we ensure programmers won’t break security?
• Bad news: **secure micro-policy composition is hard!**