Cătălin Hrițcu
Inria Paris
(visiting researcher at Microsoft until end of November)
(member of Everest expedition)

https://secure-compilation.github.io/
it’s all relative 😊

Cătălin Hrițcu
Inria Paris
(visiting researcher at Microsoft until end of November)
(member of Everest expedition)

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

•

g  tttj  aggg
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!

• ตที่ อา จ าที่ จ ท่ ง
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!

• 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!

• ăță ăĵă
  – 2015: Intel Memory Protection Extensions (MPX)
    and Intel Software Guard Extensions (SGX)

• ăță ăň ăĵă
  – 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
• 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

- 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++
  - including buffer overflows, checks too expensive
  - compilers optimize aggressively assuming undefined behavior will simply not happen
- Just write secure code ... all of it

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

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?

• ũ ũ (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++
Teasing out 2 different problems

• 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
     - all high-level safety guarantees lost
software-defined, hardware-accelerated, tag-based monitoring
software-defined, hardware-accelerated, tag-based monitoring
software-defined, hardware-accelerated, tag-based monitoring

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“store r0 r1”
software-defined, hardware-accelerated, tag-based monitoring

[Diagram showing processes and memory access with labels and instructions]

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

software monitor’s decision is hardware cached
software-defined, hardware-accelerated, tag-based monitoring

"store r0 r1"

(e.g. out of bounds write)
• 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!
• **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!
• information flow control (IFC) [POPL’14]
• 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)
• monitor self-protection
• protected compartments
• dynamic sealing
• heap memory safety
• code-data separation
• control-flow integrity (CFI)
• taint tracking
• ...

[POPL’14]
Way beyond MPX, SGX, SSM, etc
• information flow control (IFC)
• 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]

[POPL’14]
- 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

- Cătălin Hrițcu & Yannis Juglaret
- UPenn: André DeHon, Benjamin Pierce, Arthur Azevedo de Amorim, Nick Roessler
- Portland State: Andrew Tolmach
- MIT: Howie Shrobe, Stelios Sidiroglou-Douskos
- Industry: Draper Labs, Bluespec Inc

Formal methods & architecture
Micro-Policies team

- $\text{FY} \quad \& \quad g \quad \text{tn} \quad \& \quad \text{FY}$
- $\text{tn} \quad \text{FY}$:
  - *Inria Paris*: Cătălin Hriţcu $\quad j \quad g$
    (until recently $j \quad j \quad g \quad \text{tn} \quad , \quad A \quad g \quad j$)
  - *UPenn*: $j \quad j \quad j \quad , \quad A \quad j \quad \text{FY} \quad g$
    $\text{tn} \quad \text{FY} \quad \text{FY} \quad g$
  - *Portland State*: $j \quad \text{tnP} \quad \text{FY}$
  - *MIT*: $\text{ttg} \quad a$
    $g \quad g \quad g \quad \text{tn} \quad \text{tn}$
  - *Industry*: $a \quad A \quad \text{tn} \quad j$
- $\text{FY}$
Use micro-policies to build formal secure compilers for realistic programming languages.
Use micro-policies to build secure compilers for realistic programming languages – C with protected components and memory safety
Use micro-policies to build

- C with protected components and memory safety
- 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

correctness (e.g. CompCert)

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

full abstraction

low-level attacker

source target compiler program behavior program behavior compiler correctness (e.g. CompCert)

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

(e.g. CompCert)
Formally verify full abstraction

high-level attacker

low-level attacker

source target

compiler program behavior (e.g. CompCert)

correctness (e.g. arbitrary machine code)

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

Full abstraction

High-level attacker

Low-level attacker

Source

Target

Compiler

Program behavior

Program behavior

Compiler

Correctness (e.g. CompCert)

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

No extra power

Protected e.g. arbitrary machine code
Formally verify:

**full abstraction**

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

**A j g:**

- forget about compiler chain (linker, loader, runtime system)
- forget that libraries are written in a lower-level language

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

full abstraction

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

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

(e.g. CompCert)

not efficiently achievable today

A j g:  φj  t =  g  j  j  g

forget about compiler chain (linker, loader, runtime system)

forget that libraries are written in a lower-level language
Fully abstract compilation, definition

∃ g
∈ j j

∃ g
∈ j j
Fully abstract compilation, definition

∃ th

\[ g \]
\[ \bar{g} \]
\[ \bar{g} \]
\[ j j \]

≈

\[ j \]
\[ \bar{g} \]
\[ \bar{g} \]
\[ j j \]
Fully abstract compilation, definition

∃

∃

∃

∃

g

ëgyj j j

ëgyj j j

ëgyj j j

ëgyj j j

j

ëgyj j j

ëgyj j j

ëgyj j j

ëgyj j j

=>

gh

ëgyj j j

ëgyj j j

ëgyj j j

ëgyj j j

≈

≈

≈

≈
Fully abstract compilation, definition
j  j
(OCaml/F# + verification)

j  j
+ memory safety
+ components
SECOMP: achieving full abstraction at scale

miTLS* (OCaml/F# + verification)

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

- miTLS
- KremSec
- memory safe C component
- F* language (OCaml/F# + verification)
- C language + memory safety + components
SECOMP: achieving full abstraction at scale

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

- j .portal
  - memory safety
  - components

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

miTLS*

OCaml/F# + verification

+ memory safety
+ components

(RISC-V + micro-policies)

memory safe C component
SECOMP: achieving full abstraction at scale

- miTLS
- CompSec + KremSec
- memory safe C component
- legacy C component
- ASM component

- F* language (OCaml/F# + verification)
- C language + memory safety + components
- ASM language (RISC-V + micro-policies)
SECOMP: achieving full abstraction at scale

- miTLS*: protecting component boundaries
- CompSec + KremSec: memory safe C component
- Legacy C component
- ASM component

F* language (OCaml/F# + verification)

C language + memory safety + components

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

memory safe C component

legacy C component

ASM component

protecting component boundaries

j ∈ (OCaml/F# + verification)

j ∈ + memory safety + components

j ∈ (RISC-V + micro-policies)
SECOMP: achieving full abstraction at scale

miTLS* (OCaml/F# + verification)

 CompSec + memory safety + components

ASM component

C language + memory safety + components

ASM language (RISC-V + micro-policies)

Protecting component boundaries
SECOMP: achieving full abstraction at scale

**miTLS**

CompSec + KremSec

memory safe C component

legacy C component

ASM component

F* language (OCaml/F# + verification)

C language + memory safety + components

ASM language (RISC-V + micro-policies)

stronger connection to Everest expedition

protecting higher-level abstractions

protecting component boundaries
• Add mutually distrustful components to C – interacting only via g j j j
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

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

• Extending full abs. to mutual distrust + unsafe source
Towards a Fully Abstract Compiler Using Micro-policies, Juglaret et al, TR 2015
Compartmentalization micro-policy

Towards a Fully Abstract Compiler Using Micro-policies, Juglaret et al, TR 2015
Compartmentalization micro-policy

Towards a Fully Abstract Compiler Using Micro-policies, Juglaret et al, TR 2015
Compartmentalization micro-policy

Towards a Fully Abstract Compiler Using Micro-policies, Juglaret et al, TR 2015
Compartmentalization micro-policy

Compartmentalization micro-policy

memory

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

registers

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

linear return capability

@\( \text{Ret n} \)
@\( (n+1) \)

\( i \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
\( j \)
Compartmentalization micro-policy

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

memory

registers

linear return capability

C_1

C_2

pc
ra
rm

@Ret n

(n+1)

@Ret n
Compartmentalization micro-policy

\[
\begin{align*}
\text{memory} & \quad \text{registers} \\
\text{Jal } r & \quad \text{pc} \\
\ldots & \quad r_a \\
\ldots & \quad r_m \\
\ldots@\text{EntryPoint} & \quad @(n+1) \\
\text{Store } r_a \rightarrow \star r_m & \\
\ldots & \\
\text{Load } \star r_m \rightarrow r_a & \\
\text{Jump } r_a & \\
\end{align*}
\]

\[\text{g i g j at most one return capability per call stack level}\]
Compartmentalization micro-policy

```plaintext
Jal r
...
...
...@EntryPoint
Store ra \rightarrow \star rm
...
Load \star rm \rightarrow ra
Jump ra

\@Ret n

\text{@Ret n}

\text{linear return capability}

\text{@Ret n}

\text{\gsub i \gsub j}

\text{at most one return capability per call stack level}

\text{\gsub i \gsub j}
```

\[ \text{C}_1 \]

\[ \text{C}_2 \]
Compartmentalization micro-policy

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

- registers
  - \( \star \)
  - \( r_m \rightarrow r_a \)
  - \( \star \)
  - \( pc \)
  - \( r_a \)
  - \( r_m \)

- invariant:
  - at most one return capability per call stack level

\( @\) EntryPoint

\( \star \)
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]
- 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 (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 (because F* allows and tracks effects, as opposed to Coq)

• Limits of purely-dynamic enforcement

  - functional purity, termination, relational reasoning
• Protecting higher-level 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
  - fully abstract compilation of F* to ML trivial for ML interfaces (because F* allows and tracks effects, as opposed to Coq)
SECOMP focused on dynamic enforcement

- \( \hat{a} \) \( \hat{y} \) \( a \) \( g \) \( j \) \( t \) \( t \)
- \( g \) \( g \) \( j \)
  - \( \hat{y} \) \( g \) \( t \) \( g \) \( t \)
  - 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 by removing spurious checks, e.g., turn off pointer checking for a statically memory safe component that never sends or receives pointers
- improve transparency by 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.
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
- 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

  – Bringing together communities (hardware, systems, security, languages, verification, …)
• 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?

• $\mathfrak{g} \gg \mathfrak{g}$
  – secure compartmentalizing compilation (SCC)
  – preservation of hyper-safety properties [Garg et al.]

• $\mathfrak{g} \prec \mathfrak{t} \gg \mathfrak{g}$ (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?
  • \( \mathcal{g} \quad \mathcal{g} \)  
    – secure compartmentalizing compilation (SCC)  
    – preservation of hyper-safety properties [Garg et al.]
  • \( \mathcal{g} \quad \mathcal{tt} \quad \mathcal{g} \) (easier to enforce!):  
    – robust compilation (integrity but no confidentiality)
  • \( \mathcal{L} \quad \mathcal{j} \quad \mathcal{g} \) :  
    – memory safety (enforcing CompCert memory model)
What secure compilation adds over compositional compiler correctness

• ů ṣ ą ṣ ā ṭتحمل j

• ɢ ɢ ɢ ɢ
  – robust compilation phrased in terms of this

• ɢ ɢ j ĝ j ĝ ĝ ĝ ĝ ĝ ĝ
  – full abstraction and preservation of hyper-safety phrased in terms of this

• j j ĝ j ĭ j j j ĝ
  – secure compartmentalizing compilation adds this
Verification and testing

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

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

• So far all secure compilation work – but one can’t verify an interesting compiler on paper
  
• SECOMP will use: Coq and F*
  
• 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
  – but one can’t verify an interesting compiler on paper
• SECOMP will use Coq and F*
  • 
    – better automation (e.g. based on SMT like in F*)
    – integrate testing and proving (QuickChick and Luck)
• 
  – devising good for full abstraction
    is a hot research topic of it’s own
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-need for vertical compiler composition
  • will put micro-policies in the hands of programmers

• Secure micro-policy composition
  • micro-policies are interfering 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?

• g
  – 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?

• \( \text{secure compartmentalizing compilation (SCC)} \)
  – preservation of hyper-safety properties [Garg et al.]

• \( \text{robust compilation (integrity but no confidentiality)} \) (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?
  
• \( \text{secure compartmentalizing compilation (SCC)} \)
  
• \( \text{preservation of hyper-safety properties} \ [\text{Garg et al.}] \)
  
• \( \text{robust compilation (integrity but no confidentiality)} \)
  
• \( \text{memory safety (enforcing CompCert memory model)} \)
Composing compilers and higher-level micro-policies

F*

SecKremlinF*

C μ

CompSec

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

\[ F^* \rightarrow F^* \text{ component} \]

\[ \text{SecKremlinF}^* \rightarrow \text{C } \mu \rightarrow \text{CompSec} \]

\[ \text{ASM (RISC-V+}\mu\text{P)} \rightarrow \text{compiled F}^* \text{ component} \rightarrow \text{SecF}^* \mu\text{Policy} \rightarrow \text{CompSec } \mu\text{Policy} \]
Composing compilers and higher-level micro-policies

F* component

SecKremlinF*

F* component

C μ

CompSec

compiled F* component

SecF* μPolicy

CompSec μPolicy

ASM (RISC-V+μP)
User-specified higher-level policies

- By composing more micro-policies we can allow

![Diagram showing the relationship between SeKremlin μPolicy, User-specified C μPolicy, CompSec μPolicy, and user-specified ASM μPolicy.](image-url)
User-specified higher-level policies

• By composing more micro-policies we can allow

\[ C \mu P \]

• Good news:

\[ \mu P \]

since tags can be tuples

```
ASM (RISC-V+\mu P)
```

```
SeKremlin \mu P
```

```
user-specified \mu P
```

```
CompSec \mu P
```

```
user-specified ASM \mu P
```
User-specified higher-level policies

• By composing more micro-policies we can allow

\[ \mu \]

\[ \text{gg, \, g, \, gg} \]

• Good news:

\[ \text{gg, \, g, \, gg, \, j, \, g} \]

since tags can be tuples

• But how do we ensure programmers won’t break security?

\[ \begin{array}{c}
\text{C } \mu \\
\text{SeKremlin } \mu \text{Policy} \\
\text{user-specified } C \mu \text{Policy} \\
\text{CompSec } \mu \text{Policy} \\
\text{user-specified ASM } \mu \text{Policy}
\end{array} \]
User-specified higher-level policies

- By composing more micro-policies we can allow:
  \( \text{μP} \)

- Good news:
  \( \text{μP} \)
  since tags can be tuples

- But how do we ensure programmers won’t break security?
- Bad news:
  \( \text{μP} \)

\[ \text{C μP} \]

\[ \text{ASM (RISC-V+μP)} \]

\[ \text{SeKremlin μPolicy} \]

\[ \text{user-specified C μPolicy} \]

\[ \text{CompSec μPolicy} \]

\[ \text{user-specified ASM μPolicy} \]