Bătălin Hrițcu

fft  b  b  M

fft

b  b  fft
Bătălin Hrițcu

fft  b  b  b  M

_______  b  b  fft _______
Computers are insecure
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...”
• 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”
• 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”
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”
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”

“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

• B
  - b
  - b
  - b

[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?

• memory safe (at a cost)

5
• memory safe (at a cost)

• useful abstractions for writing secure code:
  – GC, type abstraction, modules, immutability, ...

fft

FFT

OCaml

Java

C#

Haskell

fft b

fft b b
<table>
<thead>
<tr>
<th>Safer high-level languages?</th>
</tr>
</thead>
<tbody>
<tr>
<td>• memory safe (at a cost)</td>
</tr>
<tr>
<td>• useful abstractions for writing secure code:</td>
</tr>
<tr>
<td>– GC, type abstraction, modules, immutability, ...</td>
</tr>
<tr>
<td>• not immune to low-level attacks</td>
</tr>
<tr>
<td>– large runtime systems, in C++ for efficiency</td>
</tr>
<tr>
<td>– unsafe interoperability with low-level code</td>
</tr>
<tr>
<td>• libraries often have large parts written in C/C++</td>
</tr>
<tr>
<td>• enforcing abstractions all the way down too expensive</td>
</tr>
</tbody>
</table>

- fft
- fft
- fft
- fft
- fft
- fft

- b
- b
- b
- b
- b
- b
- b
- b
- b
- b
- b
- b

- F B
- fft
- B
- B
- B
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

```
mem[0] = "store r0 r1"
mem[2]
mem[3]
```

Diagram:

```
bb ----> "store r0 r1" ----> fft
```
Key enabler: Micro-Policies

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

```
"store r0 r1"
```
Key enabler: Micro-Policies

store r0 r1

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

store r0 r1

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

```
mem[0] = "store r0 r1"
mem[2] = tm3'
```

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

Software monitor's decision is hardware cached
Key enabler: Micro-Policies

Monitor software-defined, hardware-accelerated, tag-based monitoring to disallow policy violations (e.g., out of bounds write).

```
store r0 r1
```

Diagram:

- bb
- fft
- fft

```
<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>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<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>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tm0 ≠ tm3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tm1 =</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<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>
```

Diagram arrows indicate flow and relationships between components.
Micro-policies are cool!

Low level + fine grained: unbounded per-word metadata, checked & propagated on each instruction.
• 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

Micro-policies are cool!
• information flow control (IFC) [POPL'14]
- 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]
• 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
- information flow control (IFC)
- monitor self-
  protection
- protected compartments
- dynamic sealing
- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking

---

**Expressiveness**

Verified (in Coq)

[Oakland'15]

Way beyond MPX, SGX, SSM, etc

[POPL’14]
- information flow control (IFC)
- monitor self-protection
- protected compartments
- dynamic sealing
- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking

Expressiveness

Verified (in Coq)

Evaluated (<10% runtime overhead)

[POPL’14]

Way beyond MPX, SGX, SSM, etc

[Oakland’15]

[ASPLOS’15]
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 Sidiroglou-Douskos
– Industry: Draper Labs, Bluespec Inc

Micro-Policies team
Formal methods & architecture

Current team:
- Inria Paris: Cătălin Hrițcuț, 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 Sidiroglou-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.
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

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

(compiler correctness (e.g. CompCert))

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

compound component

(especially not enough e.g. arbitrary machine code)
Formally verify:

full abstraction

14

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

full abstraction component

not enough e.g. arbitrary machine code
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

full abstraction component

not enough

no extra power

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

Benefit:
- Sound security reasoning in the source language
- Forget about the compiler chain (linker, loader, runtime system)
- Forget that libraries are written in a lower-level language
- Secure program behavior

Compiler correctness (e.g., CompCert)

The 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

secure program behavior

compiler correctness (e.g. CompCert)

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

full abstraction

component not enough

*folklore

not efficiently achievable today
Fully abstract compilation, definition

∃.

compiler
b b b

∃. compiler ≁ compiler

Diagram:

- Low-level attacker
- High-level component
- First compiled component
- Second compiled component
Fully abstract compilation, definition of high-level attacker leads to low-level attacker.

1st high-level component is compiled into 1st compiled component.

2nd high-level component is compiled into 2nd compiled component, which is different from the high-level attacker.

∃ low-level attacker, ⇒ compiler

Diagram:
- Two boxes represent high-level and low-level attackers.
- Arrows indicate the flow of compilation.
- The upward arrow indicates the relationship between the high-level and low-level attackers.
SECOMP: achieving full abstraction at scale

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

miTLS

KremSec

F* language (OCaml/F# + verification)

C language + 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)

C language + memory safety + components

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

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

CompSec

KremSec

miTLS

memory safe C component

protecting component boundaries
SECOMP: achieving full abstraction at scale

miTLS

CompSec + KremSec

memory safe

C component

protecting component boundaries

legacy C

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
- CompSec
- KremSec
- memory safe C component
- protecting component boundaries
- 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
CompSec + KremSec
memory safe C component
protecting component boundaries

legacy C
ASM component

F* language (OCaml/F# + verification)

C language + memory safety + components

ASM language (RISC-V + micro-policies)

protecting higher-level abstractions

stronger connection to Everest expedition
Protecting component boundaries

- Add mutually distrustful components to C – interacting only via strictly enforced interfaces
• Add mutually distrustful components to C—interacting only via strictly enforced interfaces
• CompSec compiler chain (based on CompCert)
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

B       B

fft       b

fft       b

b

b
• 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-

Jal

...@

EntryPoint

...@

Load

⋆ rm

ra

Jump

ra

memory

C1

C2

pc

ra

rm

@n+1

@Ret n

stores to the same

linear return capability

loads and stores to the same

component always allowed
Compartmentalization micro-
policy

Jal
r
 EntryPoint
...
...
...

Load
r
m
→
r
a

Jump
r
a

memory

C
1
C
2

@Ret
n

Store
r
a
→
⋆
r
m

linear return capability

@Ret
n
pc
r
a
r
m

@(n+1)

fft

Q

b

B

B

B

B

→ *

* →

b

Q

Q
Compartmentalization micro-policy

Jalr

...@

EntryPoint...

...

...

Load⋆rm

Jumpra

C1

C2

Storera→⋆rm

linear return capability

@Retn

invariant:
at most one return capability per call stack level

pcra@(n+1)
Compartmentalization micro-policy

Jalr...

Load \( r_m \rightarrow r_a \)

Jump \( r_a \)

Store \( r_a \rightarrow \star r_m \)

C2 \( \rightarrow \star \)

\@\text{Ret} n\invariant: at most one return capability per call stack level

pc \( r_a r_m \)\( (n+1) \)

cross-component return only allowed via return capability
∀b

∀b
Secure compartmentalizing compilation (SCC)

∀b

∀ low-level attack from compromised C

∃ high-level attack from some fully defined A

↯

↯

↯

↯

↯

↯

∀ compromise scenarios.
∀b

∀ b

∀b

∃ fft

∀ low-level attack from compromised C

∃ high-level attack from some fully defined A

∀ compromise scenarios.

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 ...
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, ... )
Looking for excellent interns, PhD students, PostDocs, starting researchers, and engineers.

We can also support outstanding candidates in the Inria permanent researcher competition.
• Is full abstraction always the right notion of secure compilation? The right attacker model?
• 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 hypersafety phrased in terms of this

• stronger notion of components and interfaces
  – secure compartmentalizing compilation adds this
• 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

B b b

fft B B
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

• 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

- b b b
- b b b fft

- b b b
- b policy's behavior can break another's guarantees
- fftb fft fft B b
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

F* component

SecKremlinF

compiled F*

CompSec

C

μ

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

SecF* \*μ Policy CompSec + μ Policy F* component SecKremlin F* compiled F* component

\( B \quad fft \quad ffft b \quad b \quad b \)

Q B μ
To compose compilers:

1. Higher-level micro-policies
2. Composing micro-policies
By composing more micro-policies we can allow user-specified micro-policies for C$^3_2$C$^3_2$μP$\mu$P ASM (RISC-V+μP) SeKremlin μPolicy user-specified CμPolicy CompSec μPolicy user-specified ASM μPolicy.
• 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.

\[ b \quad \text{fft} \quad b \quad b \quad b \]

\[ F \quad b \quad \text{fft} \quad b \]

\[ \begin{align*}
\text{Q} & \quad B & \quad \mu \\
& \quad & \\
& \quad & \\
& \quad B & \quad \mu & \quad b \\
& \quad B & \quad \mu & \quad b \\
& \quad B & \quad b & \quad B & \quad \mu & \quad b \\
& \quad b & \quad b & \quad b & \quad b & \quad b \\
\end{align*} \]
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!