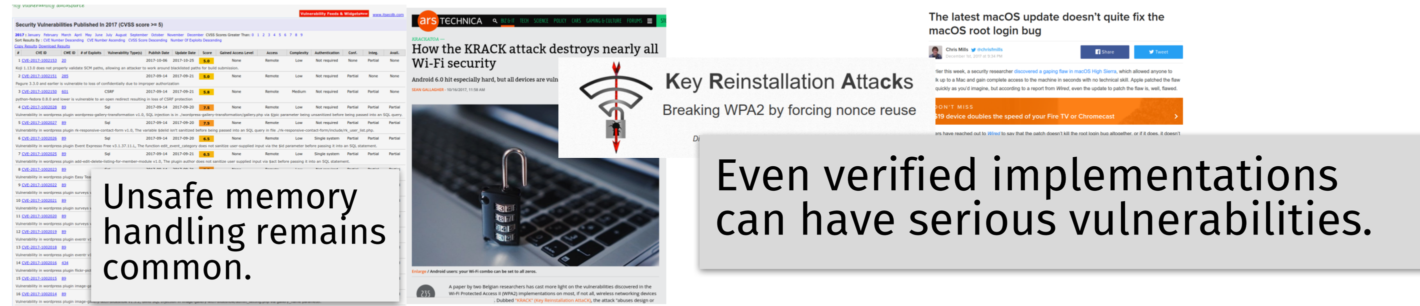
Software Fault Isolation for Robust Compilation

Ana Nora Evans, University of Virginia and INRIA Paris*

Is Software Secure?



Unsafe memory handling remains common.

Even verified implementations can have serious vulnerabilities.

Verification vs Testing

Can they coexist? Is testing still necessary?

Contributions

- Demonstrated that robust compilation can be realized on a RISC processor
- Targeted random generation of low-level code
- Novel application of property based testing to compilers and safety properties of generated code

Robust Compilation

Goals:

1. Allow reasoning about safety properties at the source level.
2. Limit the potential damage of corrupt (low-level) libraries.

A low-level compromised component cannot cause more harm than a source level one could.

Implementation:

- Proof-of-concept two-pass compiler
- Galina with Coq proofs for source to intermediate pass
- One back-end using Software Fault Isolation (presented here), another using hardware tags

Formal Definition (Source to Intermediate):
$\forall P\ C_T.\ C_T \bowtie (P\downarrow)\Downarrow t \Rightarrow \exists C_S\ t'.C_S \bowtie P \Downarrow t' \wedge t' \preccurlyeq_P t$

Memory unsafe *source* language with undefined behavior, enriched with a notion of **component** with the following constraints:

- A component can write only in its own memory.
- Each component defines an interface
  - A list of procedures others can call
  - A list of procedures it can call
- Execution can be transferred to a component only
  - By calls allowed by interface
  - Returns from cross-component calls

Target

- Load-store RISC machine
- Infinite memory
- No specialized hardware for component protection.

S ⋈ P C1 C2 C3 — Produces trace in intermediate semantics → t

∃

C_a ⋈ P C1 C2 C3 — Compiles to → Target Program — Produces trace in target semantics → t'

$\preccurlyeq_P$

Formal Definition (Intermediate to Target):
$\forall P\ C_a.\ (C_a \bowtie P)\downarrow \Downarrow t \Rightarrow \exists S\ t'.(S \bowtie P)\Downarrow t' \wedge t' \preccurlyeq_P t$

Guglielmo Fachini, Cătălin Hriţcu, Marco Stronati, Ana Nora Evans, Théo Laurent, Arthur Azevedo de Amorim, Benjamin C. Pierce, Andrew Tolmach. *Formally Secure Compilation of Unsafe Low-Level Components*. (PriSC 2018).

Software Fault Isolation

Compiler transformation to prevent:
1. Unsafe memory writes
2. Unsafe cross-component jumps

Memory Layout

Reserved (Code)	Component 1 (Code)	Component 2 (Code)	Component 3 (Code)	Protected Stack	Component 1 (Data)	Component 2 (Data)	Component 3 (Data)
Init Code	Slot 0	Slot 0	Slot 0	Slot 1	Slot 1	Slot 1	Slot 1
Unused	Slot 2	Slot 2	Slot 2	Slot 3	Slot 3	Slot 3	Slot 3
Unused	Slot 4	Slot 4	Slot 4	Slot 5	Slot 5	Slot 5	Slot 5
...	...	...	...	...	...	...	...

Transformations Examples

(component,block,offset)

Slot	Component	Offset
Unbounded	*n* bits	*s* bits

```
store *rp rs  {  RD ← rp &        11111 | 0000 | 11111111
                 RD ← RD |        00001 | cid  | 00000000
                 store *RD rs

jmp *r        {  RT← r &          11110 | 0000 | 111 | 000
                 RT← RT |         00000 | cid  | 00000000
                 jmp *RT
```

Execution continues with a corrupt address inside the current component!

- Reserved registers: **RD**, **RT**, the constants above (masks).
- RD, RT set to proper values on component change.

Cross-Component Call Stack

	Intermediate Code		Target Code
Caller Component	`call P`		`jal P'`

Callee Component	P		`*halt`
			 `P': push ra`
			 P

	`return`		`*pop ra` `jump ra`

`jal P'` is not masked! Can jump to an unaligned address.

The **halt** prevents stack corruption by preparing an address in **ra** and jumping to **P'**.

* means aligned address.

Protection of cross-component stack:

- Writes only in the data slots of the component prevent:
  - Code injected only in data slots
  - Protected stack smashing
- Execution from code slots only prevents:
  - Execution of any possible injected code
- Alignment and the halt guard prevent ROP

Internal component stack:

- Managed by the source to intermediate pass
- Stored in the component's memory
- Protected from other component
- Not protect from itself

Limitations:

- only static interfaces
- no system calls
- no compiler optimizations.

Related Work:

1. Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. *Efficient Software-based Fault Isolation.* (SOSP 1993).
2. Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. *RockSalt: Better, Faster, Stronger SFI for the x86.* (PLDI 2012).
3. Martin Abadi, Mihai Budai, Ulfar Erlingsson, Jay Ligatti. 2009. *Control-flow integrity principles, implementations, and applications.* (ACM TISS 2013).

Property Based Testing

Generators of intermediate programs:

- Used frequency combinators to increase the likelihood of tested behavior
- Generated valid intermediate memory
- Generated groups of instructions to avoid undesirable undefined behaviors like:
  `store (random int) (random register)`
  `bnz (reg from const) (random label)`
- Generated desired undefined behaviors, for example:
  `store (random address register) (random register)`

Compiler correctness

- Used CompCert definition based on traces of cross-component calls and returns
- All undefined behaviors allowed, thus the target program may produce longer traces.
- Discard programs that do not terminate in a maximum number of steps.
- Many programs with empty trace.

Randomly Generated Intermediate Program — Run in Intermediate Semantics → Intermediate Trace and Execution Result

Compile ↓ Target Program — Run in Target Semantics ↓ Target Trace, Execution Result, and Execution Log → QuickChick Checker → Passed / Failed / Discard

Target Level Tests:

- Generated a complete machine state including registers and memory
- Tested a step in the relational semantics is equivalent with a step in the RISC machine simulator.
- Proof of decidability of the step relation as complicated as a proof of the theorem itself

Shrinker:

- Build call graph
- Up to a maximum depth either
  - Replace some calls in with Nop
  - Shrink called procedures

Related Work:

1. John Hughes. *QuickCheck Testing for Fun and Profit.* (PADL'07).
2. Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin Pierce. *Foundational Property-Based Testing.* (ITP 2015).
3. Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, Leonidas Lampropoulos. *Testing noninterference, quickly.* (ICFP 2013).
4. Benjamin C. Pierce, Leonidas Lampropoulos, Zoe Paraskevopoulou. *Generating Good Generators for Inductive Relations.* (POPL 2018).

Invariant	Correctness Condition	Information Logged	Discarded Tests
Component **writes** - inside its data memory - at the top of the protected stack.	- Address and program counter in the same component - Address is the top of the protected stack	- Program counter - Store address - Top of the protected stack	Intermediate programs that fail to execute in intermediate semantics with errors unrelated to data memory access.
Jumps - within component - at addresses stored at the top of the protected stack.	- Program counter and the target address are in the same component - The target address is exactly the same as the one stored at the top of the protected stack.	- Program counter - Value of target register - Value at the top of the protected stack	Intermediate programs that fail to execute in intermediate semantics with errors unrelated to execution transfer.
Cross-component call stack is safe.	The LIFO policy is respected.	- Program counter - Top of the stack register - Operation type (push/pop) with argument.	Intermediate programs that fail to initialize correctly in the intermediate semantics.
Compiler Correctness	The intermediate trace is a sublist of the target trace.	- Intermediate trace - Target trace	Intermediate programs that fail to initialize correctly and programs that do not terminate in a maximum number of step in either intermediate or target.

Research Questions

1. Can property based testing be used to test safety properties of a program?
   Yes, if the safety properties are formulated in executable form.
2. Do randomly generated programs test the desired property?
   Mostly (see right).
3. Is testing effective in finding the implementation errors?
   Testing found errors in the compiler as well as in the testing framework itself. Future work: prove the properties in Coq.
4. What are the limitations of testing versus proofs and relational form?
   a. *Infinite loops and non-terminating programs (compiler correctness test is not complete).*
   b. *Existential quantifiers*

Future Work

Write and test the semantics of a real RISC machine (e.g., Atmel AtTiny 85 microcontroller).

Results

Store Tests

Other 8.0% (4856); Halted 12.0% (7261); Out Of Fuel 1.5%; Undefined 77.8% (46950)

83% at least one internal store

Jump Test

Discarded 5.7% (3620); Other 10.5% (6703); Halted 11.9% (7580); Out Of Fuel 1.6%; Undefined 70.3% (44699)

84% at least one internal jump

Stack Protection Test

Discarded 5.3% (3367); Bnz Undef B... 9.7% (6078); Jmp Undef 9.7% (6096); Halted 38.2% (24049); Out Of Fuel 37.1% (23347)

52% at least two stack operations

Compiler Correctess Test

Discarded 30.0% (21106); EmptyStack 2.7%; NotIntInReg 4.6% (3220); MissingBlock 1.6%; JumpNotAddre... 11.7% (8264); Halted 49.0% (34488)

10% non-empty trace

Test Type	Avg dynamic instructions	Avg static instructions
Store	58	51
Jump	31	28.7
Stack	69	52

* Work was partially performed while a visiting PhD student at **INRIA Paris** in Summer of 2017, on the **ERC SECOMP** Project.

Advisors:
Cătălin Hriţcu
Mary Lou Soffa
Marco Stronati