

# Binsec/ReISE

Efficient Constant-Time Analysis of Binary-Level Code with  
Relational Symbolic Execution

---

*Author:*

**Lesly-Ann Daniel**

CEA LIST

Oct 2018 - Oct 2021

*Supervisors:*

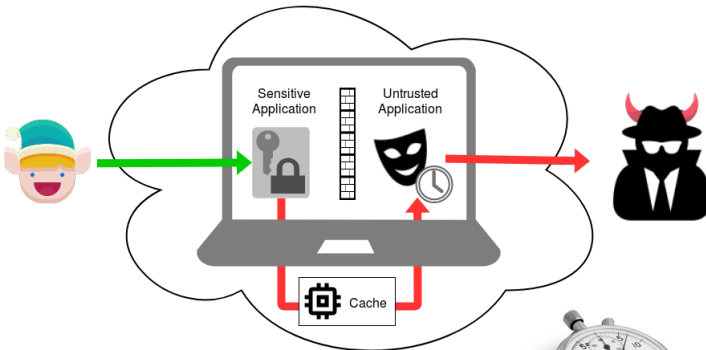
**Sébastien Bardin**

CEA LIST

**Tamara Rezk**

INRIA Sophia Antipolis

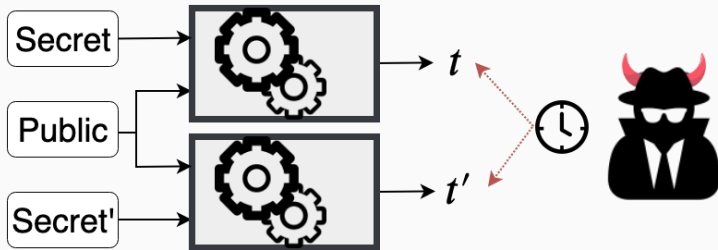
# Context: We Want to Protect our Secrets



- Confidentiality & Integrity
- Constant-time crypto.
- Secret erasure
- Spectre attacks



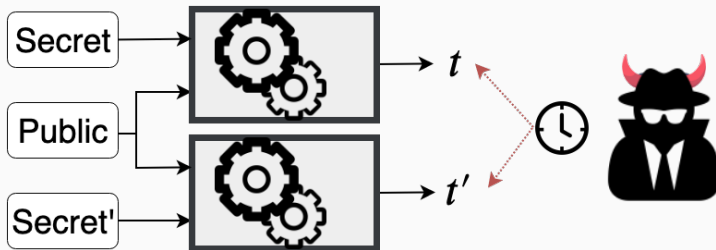
# Constant-Time Programming (CT)



What can influence time  $t$ ?

- Control flow
- Address of memory accesses (cache)

# Constant-Time Programming (CT)



What can influence time  $t$ ?

- Control flow
- Address of memory accesses (cache)

CT is a property of 2 execution traces.

## CT is not a regular safety property (2-hypersafety)

- Standard tools do not apply
- Reduction to safety with self-composition does not scale [1].

## CT is generally not preserved by compiler

- $c = (x < y) - 1$  compiled to a conditional jump?
- Depends on compiler options and optimizations [2].

*Requires tools for 2-hypersafety & binary-level reasoning.*

---

[1] “Secure information flow as a safety problem”, Terauchi and Aiken 2005

[2] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

# Problem

CT verification tools target:

- source code [1,2]
- LLVM code [3,4]

Binary-level tools:

- Dynamic analysis (sacrifice BV) [5]
- Sound over-approx. (sacrifice BF) [6]
- Do not scale [7]

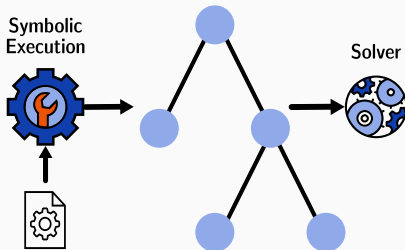
***Goal: Design Efficient BF & BV Tool for CT at Binary-Level***

- 
- [1] "Formal verification of side-channel countermeasures using self-composition", Bacelar Almeida et al. 2013
  - [2] "Verifying Constant-Time Implementations by Abstract Interpretation", Blazy, Pichardie, and Trieu 2017
  - [3] "Verifying Constant-Time Implementations.", Almeida et al. 2016
  - [4] "CaSym: Cache aware symbolic execution for side channel detection and mitigation", Brotzman et al. 2019
  - [5] "CacheD: Identifying Cache-Based Timing Channels in Production Software", Wang et al. 2017
  - [6] "Rigorous analysis of software countermeasures against cache attacks", Doychev and Köpf 2017
  - [7] "Verifying information flow properties of firmware using symbolic execution", Subramanyan et al. 2016

# Bug Finding? Try Symbolic Execution

## Symbolic Execution

- Leading formal method for BF
- Precise (no false alarm)
- Scales better than other semantic analysis
- Widely used in intensive testing and security analysis
- Can also be used for bounded verification



# Key Insights: Adapt SE for CT

Goal: Adapt Symbolic Execution for CT		
Bug Finding	Bounded Verification	Scalability

**Relational SE:** 2 programs in the same SE instance [1,2]

- Formula sharing
- Spared checks

For source code  
Do not scale at binary-level

**Binary-Level RelSE:** Dedicated optims

- On-the-fly simplification for binary-level reasoning
- Untainting
- Fault-packing

New

---

[1] "Shadow of a doubt", Palikareva, Kuchta, and Cadar 2016

[2] "Relational Symbolic Execution", Farina, Chong, and Gaboardi 2017



**Dedicated optims for CT analysis at Binary-Level**

**Binsec/Rel: First efficient BF & BV tool for CT**

**Large Scale Experiments**

**Extension of study on CT preservation by compilers [4]**

## **Dedicated optims for CT analysis at Binary-Level**

- Existing ones: relies on RelSE to improve sharing
- New ones for binary: on-the-fly binary-level simplification
- New ones for CT analysis: untainting & fault-packing

## **Binsec/Rel: First efficient BF & BV tool for CT**

## **Large Scale Experiments**

## **Extension of study on CT preservation by compilers [4]**

**Dedicated optims for CT analysis at Binary-Level**

**Binsec/Rel: First efficient BF & BV tool for CT**

- Extensive experimental evaluation (338 samples)
- $700\times$  speedup compared to RelSE
- $1.8\times$  overhead compared to SE

**Large Scale Experiments**

**Extension of study on CT preservation by compilers [4]**

**Dedicated optims for CT analysis at Binary-Level**

**Binsec/Rel: First efficient BF & BV tool for CT**

**Large Scale Experiments**

- New proofs on binary previously done on C/LLVM/F\* [1,2,3]
- Replay of known bugs (e.g. Lucky13)

**Extension of study on CT preservation by compilers [4]**

---

[1] “Verifying Constant-Time Implementations.”, Almeida et al. 2016

[2] “Verifying Constant-Time Implementations by Abstract Interpretation”, Blazy, Pichardie, and Trieu 2017

[3] “HACL\*”, Zinzindohoué et al. 2017

**Dedicated optims for CT analysis at Binary-Level**

**Binsec/Rel: First efficient BF & BV tool for CT**

**Large Scale Experiments**

**Extension of study on CT preservation by compilers [4]**

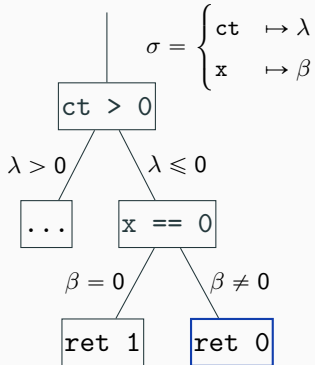
- Automatization
- More implementations
- gcc compiler + newer version of clang
- ARM binaries
- Discover new bugs out of reach of previous tools for LLVM

---

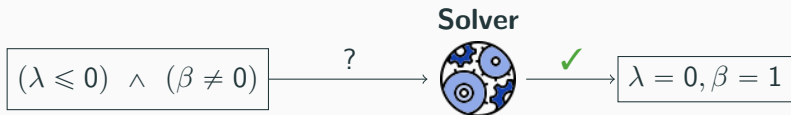
[4] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

# Background: Symbolic Execution

```
1  int is_zero(  
2      uint32 ct, // public  
3      uint32 x){ // private  
4  if (ct > 0) {  
5      y = ~x & (x-1);  
6      return y >> 31;  
7  } else {  
8      if (x == 0) return 1;  
9      else return 0;  
10 }
```

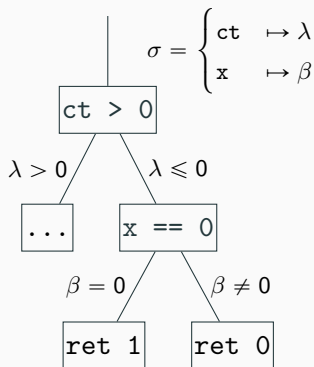


**Question:** How to reach line 9?



# Background: SE & Self-Composition for CT

```
1  int is_zero(  
2      uint32 ct, // public  
3      uint32 x){ // private  
4  if (ct > 0) {  
5      y = ~x & (x-1);  
6      return y >> 31;  
7  } else {  
8      if (x == 0) return 1;  
9      else return 0;  
10 } }
```



**Question:** Can “x” leak w.r.t. CT policy?

**Jump 4.**  $((\lambda = \lambda') \wedge (\lambda > 0 \neq \lambda' > 0))?$   $\rightarrow$  **unsat** ✓

**Jump 8.**  $\left( (\lambda = \lambda') \wedge (\lambda \leq 0) \wedge (\lambda' \leq 0) \right. \\ \left. \wedge (\beta = 0 \neq \beta' = 0) \right)? \rightarrow$  **sat**  $\begin{matrix} (\lambda = 0, \beta = 1) \\ (\lambda' = 0, \beta' = 0) \end{matrix}$

## Problem: SE & Self-Composition for CT

**Self-Composition:** no sharing between both executions

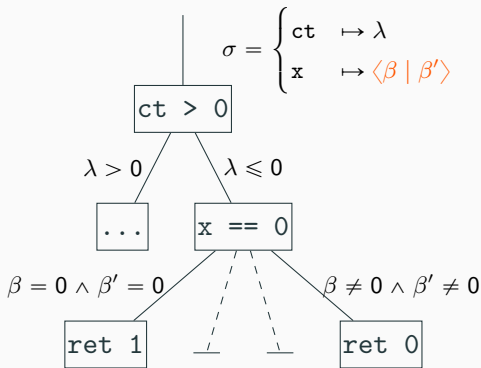
- size of queries  $\times 2$
- does not keep track of secret dependencies
- high number of insecurity queries

*Symbolic-Execution & Self-Composition for CT does not scale.*



# Relational SE for CT

```
1  int is_zero(  
2      uint32 ct, // public  
3      uint32 x){ // private  
4  if (ct > 0) {  
5      y = ~x & (x-1);  
6      return y >> 31;  
7  } else {  
8      if (x == 0) return 1;  
9      else return 0;  
10 }}}
```



**Question:** Can “x” leak w.r.t. CT policy?

**Jump 4.** Spared query  $\rightarrow \checkmark$

**Jump 8.**  $((\lambda \leq 0) \wedge (\beta = 0 \neq \beta' = 0))?$   $\rightarrow$  **sat**  $(\lambda = 0, \beta = 1, \beta' = 0)$

# Challenge: Binary-Level Reasoning

**Relational SE:** sharing via relational expressions

- keeps track of secret dependencies
- $\searrow$  # insecurity queries
- $\searrow$  size of queries
- scales better

# Challenge: Binary-Level Reasoning

**Relational SE:** sharing via relational expressions

- keeps track of secret dependencies
- $\searrow$  # insecurity queries
- $\searrow$  size of queries
- scales better

**Problem** Does not scale for binary analysis

- Memory is represented as a symbolic array variable
- Duplicated at the beginning of RelSE
- Duplicate all the load operations

# Dedicated Simplifications for Binary-Level ReSE

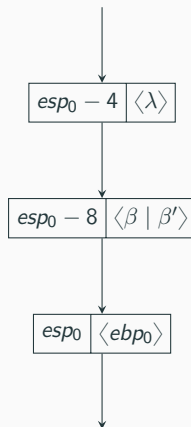
## Problem

ReSE + Binary = Duplicated Memory

**FlyRow: on-the-fly read-over-write**

- Build on *read-over-write* [1]
- Simplify load operations on-the-fly
- Relational expressions in the memory

Memory as the  
history of stores



---

[1] "Arrays Made Simpler", Farinier et al. 2018

# Dedicated Simplifications for CT Analysis

## Untainting

Solver says  $\beta \neq \beta'$  is UNSAT  $\implies$  Replace  $\langle \beta \mid \beta' \rangle$  by  $\langle \beta \rangle$  in SE.

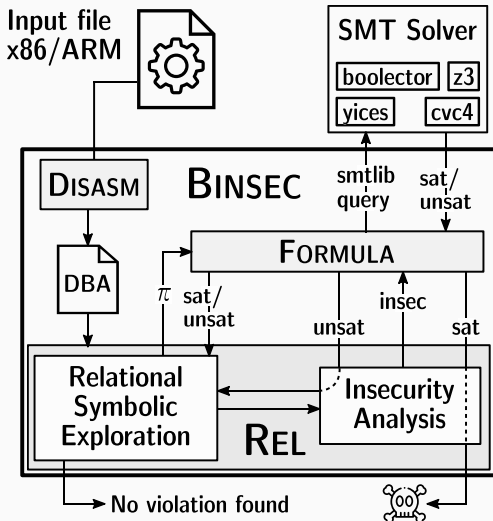
- + Track secret-dependencies more precisely
- + Spare insecurity queries

## Fault-Packing

Pack insecurity queries along the analysis and send them at the end of a basic-block.

- + Reduces number of queries
- + Useful for CT (lot of insecurity queries)
- Precision loss: violations at basic-block lvl

# Implementation



## Overall:

- Part of BINSEC
- $\simeq 3.5k$  lines of Ocaml
- IR: DBA
- Input: x86 / ARM binary

## Usability:

- Stubs for specification
- IDA plugin for visualization




## Scalability: Comparison with Standard Approaches

	#I	#I/s	#Q	T	⌚	✓	✗
SC	252k	3.9	170k	65473	15	282	41
RelSE	320k	5.4	97k	59316	14	283	42
BINSEC/REL	22.8M	3861	3.9k	5895	0	296	42

*Total on 338 cryptographic samples (secure & insecure)*

Conclusion			
↘ ×25 #Q	↘ T	↗ ×700 #I/s	↘ ⌚

# Scalability: Performances of Optimizations

Version	#I	#I/s	#Q	T			
<b>Standard RelSE with <i>Unt</i> and <i>fp</i></b>							
<i>RelSE</i>	320k	5.4	96919	59316	14	283	42
+ <i>Unt</i>	373k	8.4	48071	44195	8	288	42
+ <i>fp</i>	391k	10.5	33929	37372	7	289	42
<b>Binsec/Rel (<i>RelSE</i> + <i>FlyRow</i> + <i>Unt</i> + <i>fp</i>)</b>							
<i>RelSE</i> + <i>FlyRow</i>	22.8M	3075	4018	7402	0	296	42
+ <i>Unt</i>	22.8M	3078	4018	7395	0	296	42
+ <i>fp</i>	22.8M	3861	3980	5895	0	296	42

- *FlyRow*: major source of improvement
- *Unt* and *fp*: positive impact on *RelSE*
- *Unt* and *fp*: modest impact on *FlyRow*



- BINSEC/REL: only  $\times 1.8$  overhead compared to our best SE
- *FlyRow* outperforms SOA *ROW* as post-processing [1]
- *FlyRow* also improves standard SE  $\#I/s \times 450$ .

---










[1] “Arrays Made Simpler”, Farinier et al. 2018

## Efficiency: Bounded-Verification

		$\sim \#l$	$\#l_u$	T	S
utility	ct-select	1015	1507	.21	29 × ✓
	ct-sort	2400	1782	.24	12 × ✓
	Hacl*	3850	90953	9.34	110 × ✓
	OpenSSL	4550	5113	.75	130 × ✓
tea	-00 & -03	540	1757	.24	2 × ✓
donna	-00 & -03	11726	12.9M	1561	2 × ✓
libsodium	salsa20 & chacha20	4344	30.5k	5.7	2 × ✓
	sha256 & sha512	21190	100.8k	11.6	2 × ✓
Hacl*	chacha20	1221	5.0k	1.0	✓
	curve25519	8522	9.4M	1110	✓
	sha256 & sha512	3292	48.6k	7.1	2 × ✓
BearSSL	aes_ct & des_ct	1039	42.0k	34.5	2 × ✓
OpenSSL	tls-rempad-patch	424	35.7k	406	✓
<b>Total</b>		64114	22.7M	3154	296 × ✓

**Conclusion:** First automatic CT-analysis at binary level

## Efficiency: Bug-Finding

		$\sim \#l$	$\#l_u$	T	CT <sub>src</sub>	S		Comment
utility	ct-select	735	767	.29	Y	21× 	21	1 new 
	ct-sort	3600	7513	13.3	Y	18× 	44	2 new 
BearSSL	aes_big	375	873	1574	N		32	-
	des_tab	365	10421	9.4	N		8	-
OpenSSL	tls-remove-pad-lucky13	950	11372	2574	N		5	-
<b>Total</b>		6025	30946	4172	-	42× 	110	-

**Conclusion:** First automatic CT-analysis at binary level

# Effect of compiler optimizations on CT (see [1])

## Extension of study on CT preservation by compilers [1]

- Automatization
- 29 new functions
- add 2 gcc compiler + clang v7.1 for x86
- ARM binaries

---

[1] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

# Effect of compiler optimizations on CT (extension of [1])

	cl-3.0		cl-3.9		cl-7.1		gcc-5.4		gcc-8.3		arm-gcc	
	00	03	00	03	00	03	00	03	00	03	00	03
ct_select_v1	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v2	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v3	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v4	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
select_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
ct_sort	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
ct_sort_mult	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
sort_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
hacl_utility (×11)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
openssl_utility (×13)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
tea_enc & dec (×2)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Shows genericity of BINSEC/REL: several compilers, and x86/ARM arch

[1] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

[2] “Verifying Constant-Time Implementations.”, Almeida et al. 2016

# Effect of compiler optimizations on CT (extension of [1])

	cl-3.0		cl-3.9		cl-7.1		gcc-5.4		gcc-8.3		arm-gcc	
	00	03	00	03	00	03	00	03	00	03	00	03
ct_select_v1	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v2	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v3	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v4	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
select_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
ct_sort	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
ct_sort_mult	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
sort_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
hacl_utility (×11)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
openssl_utility (×13)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
tea_enc & dec (×2)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

clang optimizations tend to break CT (supports [1])

[1] "What you get is what you C", Simon, Chisnall, and Anderson 2018

[2] "Verifying Constant-Time Implementations.", Almeida et al. 2016

# Effect of compiler optimizations on CT (extension of [1])

	cl-3.0		cl-3.9		cl-7.1		gcc-5.4		gcc-8.3		arm-gcc	
	00	03	00	03	00	03	00	03	00	03	00	03
ct_select_v1	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v2	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v3	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v4	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
select_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
ct_sort	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
ct_sort_mult	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
sort_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
hacl_utility (×11)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
openssl_utility (×13)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
tea_enc & dec (×2)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

newer clang versions not more likely to break CT (contradicts [1])

[1] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

[2] “Verifying Constant-Time Implementations.”, Almeida et al. 2016

# Effect of compiler optimizations on CT (extension of [1])

	cl-3.0		cl-3.9		cl-7.1		gcc-5.4		gcc-8.3		arm-gcc	
	00	03	00	03	00	03	00	03	00	03	00	03
ct_select_v1	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v2	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v3	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v4	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
select_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
ct_sort	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
ct_sort_mult	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
sort_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
hacl_utility (×11)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
openssl_utility (×13)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
tea_enc & dec (×2)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

gcc optimizations tend to remove branches (especially in ARM)

[1] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

[2] “Verifying Constant-Time Implementations.”, Almeida et al. 2016



# Effect of compiler optimizations on CT (extension of [1])

	cl-3.0		cl-3.9		cl-7.1		gcc-5.4		gcc-8.3		arm-gcc	
	00	03	00	03	00	03	00	03	00	03	00	03
ct_select_v1	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v2	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v3	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v4	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
select_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
ct_sort	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
ct_sort_mult	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
sort_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
hacl_utility (×11)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
openssl_utility (×13)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
tea_enc & dec (×2)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

ct\_select\_v1 can be compiled to insecure binary (contradict [1])

[1] "What you get is what you C", Simon, Chisnall, and Anderson 2018

[2] "Verifying Constant-Time Implementations.", Almeida et al. 2016

# Effect of compiler optimizations on CT (extension of [1])

	cl-3.0		cl-3.9		cl-7.1		gcc-5.4		gcc-8.3		arm-gcc	
	00	03	00	03	00	03	00	03	00	03	00	03
ct_select_v1	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v2	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v3	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v4	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
select_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
ct_sort	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
ct_sort_mult	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
sort_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
hacl_utility (×11)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
openssl_utility (×13)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
tea_enc & dec (×2)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

ct\_sort compiled with gcc -O0 is not secure (out of reach of ct-verif [2])

[1] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

[2] “Verifying Constant-Time Implementations.”, Almeida et al. 2016

# Effect of compiler optimizations on CT (extension of [1])

	cl-3.0		cl-3.9		cl-7.1		gcc-5.4		gcc-8.3		arm-gcc	
	00	03	00	03	00	03	00	03	00	03	00	03
ct_select_v1	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v2	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v3	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
ct_select_v4	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
select_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
ct_sort	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
ct_sort_mult	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
sort_naive (insecure)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
hacl_utility (×11)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
openssl_utility (×13)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
tea_enc & dec (×2)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

clang backend passes break ct\_sort (deemed secure by ct-verif [2])

[1] “What you get is what you C”, Simon, Chisnall, and Anderson 2018

[2] “Verifying Constant-Time Implementations.”, Almeida et al. 2016

## Conceptual Limitations

- Loop unrolling → fine for bugs but limits proofs

## Implementation Limitations

- No system calls → requires stubs
- No dynamic libraries → statically linked binaries

## Experiments

- esp is concretized (like in related work)
- No dynamic allocation → Fixed array length (keys, plaintext)

## Efficient BF/BV for CT at Binary-Level Experiments on crypto implementations

- **BF** ✓ (no over-approx) & **BV** ✓ (no under-approx)
- **Sharing for Scaling**
  - Relational SE
  - Dedicated optimizations
- **Binary-level**
  - No source code needed
  - Do not rely on compiler



- Spectre (already a prototype)
  - New properties (e.g. cache model, secret erasure, etc.)
  - General noninterference
- 

- Any idea of new properties or use cases?
-