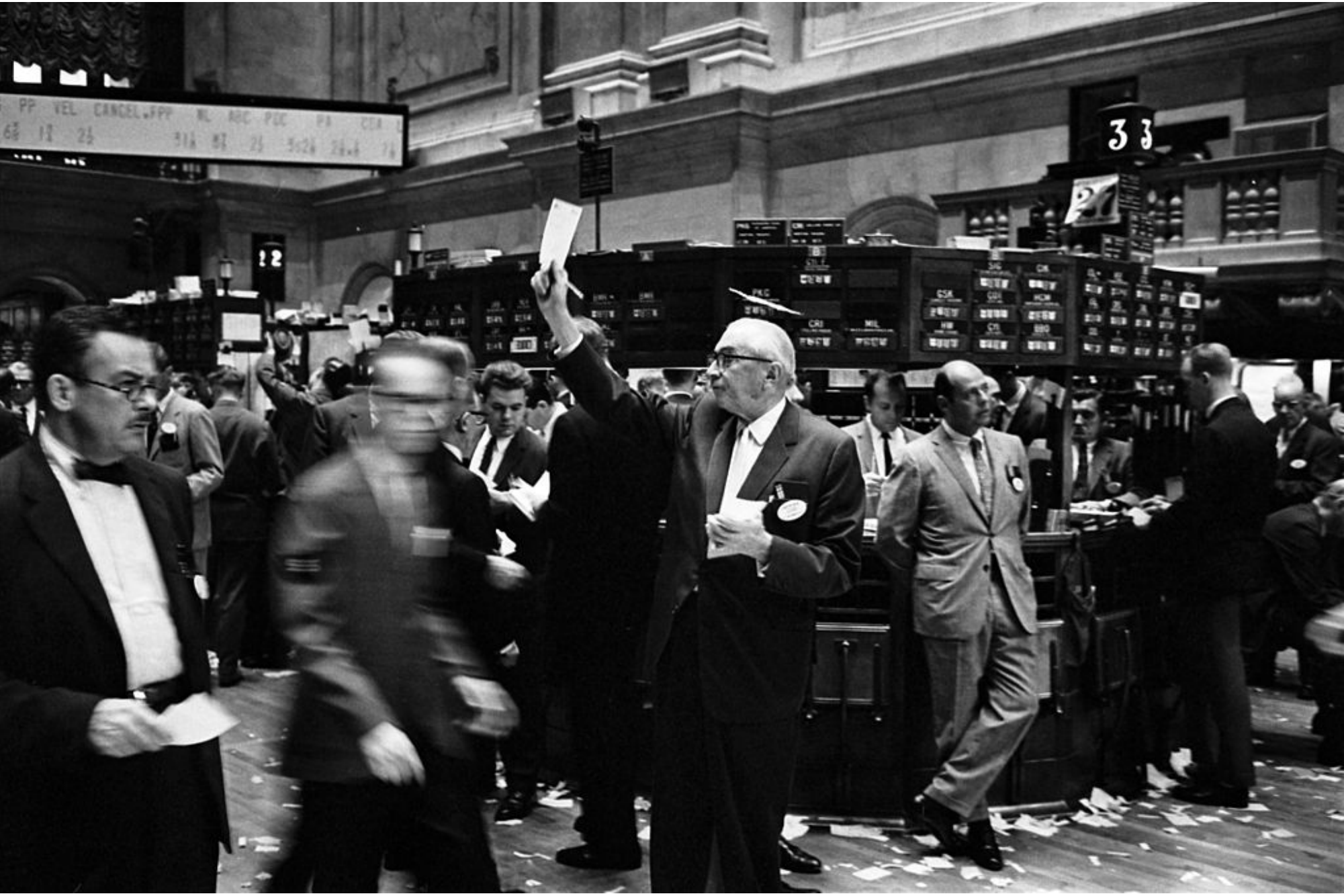
# Well-typed smart fuzzing

Thomas Braibant  Jonathan Protzenko  Gabriel Scherer

Cryptosense  Inria

ML Family Workshop


33


nCIPHER™
STATUS
CLEAR


Luna SA
SafeNet

**U.S. dept. of Justice**

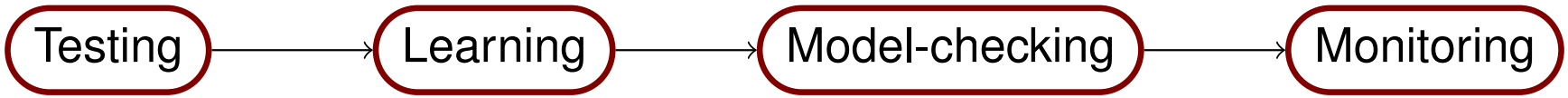
**Two heists.**

**5M$ + 40M$.**

**5+12 accounts**

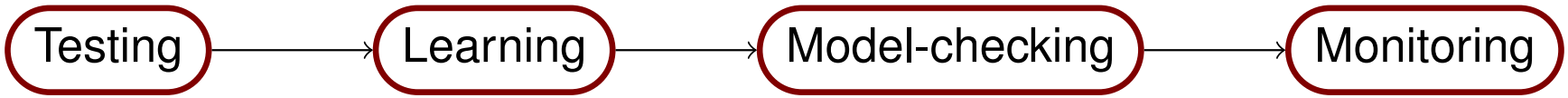
**3+10 hours**

# Cryptosense workflow




In this talk, I will focus on testing

# Cryptosense workflow




**In this talk, I will focus on testing**

# Writing test cases is a pain.

**Can I generate test cases automatically?**

YES: QuickCheck.

- A combinator library to generate test cases in Haskell

*and a host of other languages*

- Generate test cases for functions based on their types.
  E.g. `val insert: int rbt → int → int rbt`

BUT:

- What about testing the interaction of several functions?
- How to deal with abstract types?

# Articheck

**A prototype library to generate test cases for safety properties**

| QuickCheck | Articheck |
|---|---|
| Whitebox (internal) | Blackbox (external) |
| per function | at the API level |
| generate random values | combine function calls |
| use type definitions | use the interface |

# Leveraging APIs

- Types help generate good random values.
- APIs help generate values that have the right internal invariants.
- Yet. Well-typed does not mean well-formed! We are still fuzzing.

# Leveraging APIs

- Types help generate good random values.
- APIs help generate values that have the right internal invariants.
- Yet. Well-typed does not mean well-formed! We are still fuzzing.

# Leveraging APIs

- Types help generate good random values.
- APIs help generate values that have the right internal invariants.
- Yet. Well-typed does not mean well-formed! We are still fuzzing.

# Outline

1. Describing and testing APIs in a nutshell
2. Taming the combinatorial explosion
3. An industrial perspective: crypto APIs

# Describing APIs

# What's an API?

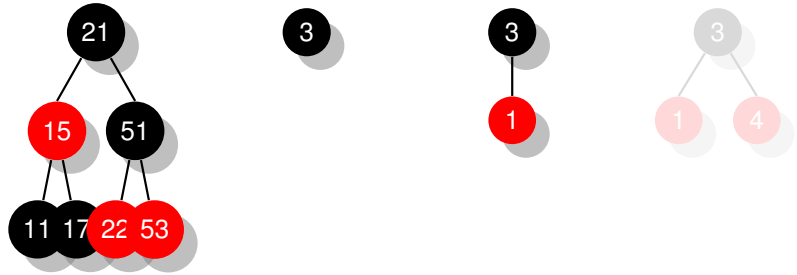
```
(* Mock Set API*)
type 'a t
val empty: 'a t
val insert: 'a → 'a t → 'a t
val delete: 'a → 'a t → 'a t
val mem: 'a → 'a t → bool
```

```
(* Mock crypto API*)
type template
type key
type text = string
val generate_key: template → key
val wrap_key: key → key → text
val encrypt: key → text → text
```

**In this talk, APIs are first-order.**

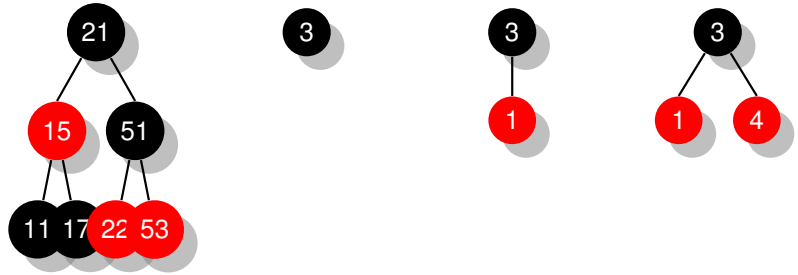
# Testing an implementation of red-black trees

```
(* Mock Set API*)
type 'a t
val empty: 'a t
val insert: 'a → 'a t → 'a t
val delete: 'a → 'a t → 'a t
val mem: 'a → 'a t → bool
```



# Testing an implementation of red-black trees

```
(* Mock Set API*)
type 'a t
val empty: 'a t
val insert: 'a → 'a t → 'a t
val delete: 'a → 'a t → 'a t
val mem: 'a → 'a t → bool
```



# A toy DSL for describing APIs

**Describing types.**

```
type 'a ty = {name: string; mutable content: 'a set; ...}
```

**Describing functions.**

```
type ('a,'b) fn =
| ret : 'a ty → ('a,'a) fn
| @→ : 'a ty → ('b,'c) fn → ('a → 'b,'c) fn
```

**Describing signatures.**

```
type elem = Elem: string * ('a,'b) fn * 'a → elem
type signature = elem list
let declare label fn f = ...
```

# A toy DSL for describing APIs

**Describing types.**

```
type 'a ty = {name: string; mutable content: 'a set; ...}
```

**Describing functions.**

```
type ('a,'b) fn =
| ret : 'a ty → ('a,'a) fn
| @→ : 'a ty → ('b,'c) fn → ('a → 'b,'c) fn
```

**Describing signatures.**

```
type elem = Elem: string * ('a,'b) fn * 'a → elem
type signature = elem list
let declare label fn f = ...
```

# A toy DSL for describing APIs

### Describing types.

```
type 'a ty = {name: string; mutable content: 'a set; ...}
```

### Describing functions.

```
type ('a,'b) fn =
| ret : 'a ty → ('a,'a) fn
| @→ : 'a ty → ('b,'c) fn → ('a → 'b,'c) fn
```

### Describing signatures.

```
type elem = Elem: string * ('a,'b) fn * 'a → elem
type signature = elem list
let declare label fn f = ...
```

# Coming back to red-black trees

```
let int_ty : int ty = ...
let t_ty : (int RBT.t) ty  = ...
let api = [
  declare "empty"  (ret t_ty) RBT.empty;
  declare "insert" (int_ty @→ t_ty @→ ret t_ty) RBT.insert;
  declare "delete" (int_ty @→ t_ty @→ ret t_ty) RBT.delete;
  declare "mem"    (int_ty @→ t_ty @→ ret bool_ty) RBT.mem;
]
```

**Many possible strategies to combine these functions.**

*depth-first, breadth-first, round-robin, random walks*

**This yields a list of inhabitants for the types:**

- used to check invariants (e.g., being a red-black tree)
- used to check correctness properties
  $\forall$ `x : t_ty`, $\forall$ `e : int_ty`, `delete x (insert x t) = t`

# Coming back to red-black trees

```
let int_ty : int ty = ...
let t_ty : (int RBT.t) ty  = ...
let api = [
  declare "empty"  (ret t_ty) RBT.empty;
  declare "insert" (int_ty @→ t_ty @→ ret t_ty) RBT.insert;
  declare "delete" (int_ty @→ t_ty @→ ret t_ty) RBT.delete;
  declare "mem"    (int_ty @→ t_ty @→ ret bool_ty) RBT.mem;
]
```

**Many possible strategies to combine these functions.**

*depth-first, breadth-first, round-robin, random walks*

**This yields a list of inhabitants for the types:**

- used to check invariants (e.g., being a red-black tree)
- used to check correctness properties
  ∀ x : t_ty, ∀ e : int_ty, delete x (insert x t) = t

# A richer DSL for types

```
type _ ty =
| Abstract: 'a abstract → 'a ty
| Sum: 'a ty * 'b ty → ('a,'b) Either.t ty
| Product: 'a ty * 'b ty → ('a * 'b) ty
| Filter: 'a ty * ('a → bool) → 'a ty
| Bijection: 'a ty * ('a → 'b) * ('b → 'a) → 'b ty
```

**Dolev-Yao style: A user/attacker can decompose sum and products**

# Field report: Crypto APIs

# Some issues

- We need to enumerate a big combinatorial space made of:
  - constants (types whose inhabitants are known, e.g. templates, key types, mechanisms)
  - variables (types populated dynamically, e.g. handles, ciphertexts)
- We want a good coverage (“smaller values” first)
- We want a reproducible behavior (i.e., avoid random testing)

# Feature #1: a library of enumerators

Low memory footprint: if possible, constant space

Efficient access: if possible, constant time for random accesses

Some bad solutions:

- Arrays: one cell per element, contiguous, ...
- Lists: one cell per element, linear time accesses, ...
- Lazy lists: idem, ...

# Quick peek

```
type 'a t = {size: int; nth: int → 'a}

val of_list     : 'a list → 'a t
val product     : 'a t → 'a t → ('a * 'b) t
val map         : ('a → 'b) → 'a t → 'b t
val append      : 'a t → 'a t → 'a t
...
val subset      : 'a t list → 'a list t t
...
val squash      : 'a t t → 'a t
val round_robin : 'a t t → 'a t
```

`subset` is paramount to generate templates (list of attributes) efficiently.

# Feature #2: a library for combinatorial enumeration

```
(* Combinators to describe the combinatorial problem *)
type 'a d
val constant: 'a Enum.t → 'a d
val variable: 'a set    → 'a d
val filter  : ('a → bool) → 'a d → 'a d
val map     : ('a → 'b) → 'a d → 'b d
val sum     : 'a d → 'b d → ('a,'b) Either.t d
val product : 'a d → 'b d → ('a * 'b) d

(* Imperative updates to the state of the exploration*)
type 'a t
val start: 'a d → 'a t
val take : 'a t → 'a option
val add  : 'a → 'a set → unit
```

- The state is roughly an enumerator and an index.
- When empty, compute an enumerator for the next elements to process
  - avoiding redundancy with what has been done already;
  - maximizing throughput

# Feature #2: a library for combinatorial enumeration

```
(* Combinators to describe the combinatorial problem *)
type 'a d
val constant: 'a Enum.t → 'a d
val variable: 'a set    → 'a d
val filter  : ('a → bool) → 'a d → 'a d
val map     : ('a → 'b) → 'a d → 'b d
val sum     : 'a d → 'b d → ('a,'b) Either.t d
val product : 'a d → 'b d → ('a * 'b) d

(* Imperative updates to the state of the exploration*)
type 'a t
val start: 'a d → 'a t
val take : 'a t → 'a option
val add  : 'a → 'a set → unit
```

- The state is roughly an enumerator and an index.
- When empty, compute an enumerator for the next elements to process
  - avoiding redundancy with what has been done already;
  - maximizing throughput

# Quick peek

Compute the effect of adding a new element $v$ to a variable $\alpha$.

$$
\begin{array}{rcl}
\delta_\alpha(k) & \triangleq & \emptyset \\
\delta_\alpha(\alpha) & \triangleq & \{v\} \\
\delta_\alpha(\beta) & \triangleq & \emptyset \\
\delta_\alpha(f\ b) & \triangleq & f(\delta_\alpha b) \\
\delta_\alpha(a \times b) & \triangleq & \delta_\alpha(b) \times c \cup b \times \delta_\alpha(c)
\end{array}
$$

# Testing an HSM

| | | | | | |
|---|---|---|---|---|---|
| C_GetTokenInfo | 1 | 1 | | 0.001 | 0 |
| C_GenerateKey | 30095 | 10705 | 19390 | 1.730 | 0 |
| C_GenerateKeyPair | 116563 | 22478 | 94085 | 100.224 | 9382739 |
| C_CreateObject (DES) | 6019 | 2141 | 3878 | 0.339 | 0 |
| C_CreateObject (DES3) | 6019 | 2141 | 3878 | 0.357 | 0 |
| C_CreateObject (AES) | 18057 | 6423 | 11634 | 1.236 | 0 |
| C_CreateObject (RSA, public) | 1161 | 577 | 584 | 0.143 | 0 |
| C_CreateObject (RSA, private) | 4091 | 1199 | 2892 | 1.122 | 0 |
| C_Encrypt (with symmetric key) | 110160 | 4716 | 105444 | 5.249 | 0 |
| C_Encrypt (with asymmetric key) | 67648 | 37730 | 29918 | 6.691 | 47 |
| C_Decrypt (with symmetric key) | 116466 | 111 | 116355 | 9.424 | 360498 |
| C_Decrypt (with asymmetric key) | 116563 | 82 | 116477 | 149.626 | 250910 |
| C_WrapKey (with symmetric key) | 116563 | 1824 | 114737 | 9.833 | 8198753 |
| C_WrapKey (with asymmetric key) | 116563 | 72862 | 43674 | 119.661 | 368429621 |
| C_GetAttributeValue (key value) | 57469 | 24736 | 32733 | 19.800 | 0 |
| C_SetAttributeValue | 116562 | 3909 | 112653 | 17.796 | 387844143 |

**$10^6$ tests in 540s. 20%overhead. 0 failures.**

# Take away

**We have a principled way to test (persistent) APIs.**

# One more thing about Cryptosense

We find bugs in HSMs.
We find logical attacks on crypto APIs.
We are recruiting.