

Kindly Bent To Free Us

Gabriel Radanne Hannes Saffrich Peter Thiemann

“high severity security bugs” in Chromium

High+, impacting stable

Security-related assert

7.1%

Other

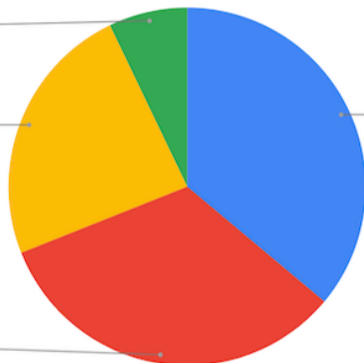
23.9%

Other memory unsafety

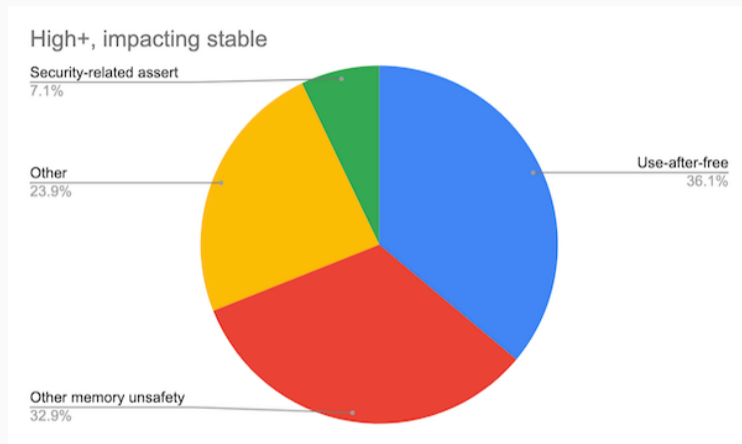
32.9%

Use-after-free

36.1%



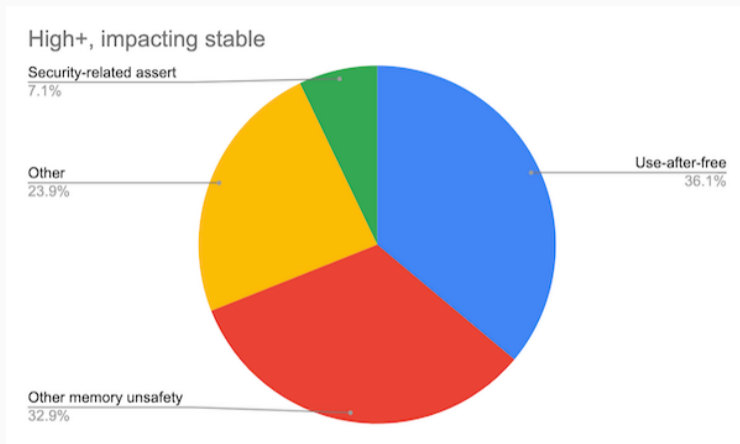
“high severity security bugs” in Chromium



Chromium is written in C/C++!

Surely these bugs don't happen in high-level typed languages.

“high severity security bugs” in Chromium



Chromium is written in C/C++!

Surely these bugs don't happen in high-level typed languages ... right?

Let's write some OCaml code

```
let gradeDB : database = Dbm.opendbm "gradeDB" ... in
Dbm.add gradeDB "math" 42;
(* ... *)
Dbm.close gradeDB;
(* ... *)
print_int (Dbm.find gradeDB "literature") (* run-time error! *)
```

```
let gradeDB : database = Dbm.opendbm "gradeDB" ... in
Dbm.add &!gradeDB "math" 42;
(* ... *)
Dbm.close gradeDB;
(* ... *)
print_int (Dbm.find &gradeDB "literature") (* x compile-time error! *)
```

```
type database : lin (* Linear kind *)
```

```
let gradeDB : database = Dbm.opendbm "gradeDB" ... in
```

```
Dbm.add &!gradeDB "math" 42;
```

```
(* ... *)
```

```
Dbm.close gradeDB;
```

```
(* ... *)
```

```
print_int (Dbm.find &gradeDB "literature") (* x compile-time error! *)
```


```
type string : un (* Unrestricted kind *)
```

Kinds determine usage

```
let gradeDB : database = Dbm.opendbm "gradeDB" ... in
Dbm.add &!gradeDB "math" 42;
(* ... *)
Dbm.close gradeDB;
(* ... *)
print_int (Dbm.find &gradeDB "literature") (* x compile-time error! *)
```

The diagram illustrates a borrow relationship between two uses of the variable `&gradeDB`. A box labeled "Borrows!" has two arrows: one pointing to the `&!gradeDB` in the `Dbm.add` line and another pointing to the `&gradeDB` in the `Dbm.find` line. This indicates that the second use borrows the value from the first use, which is a compile-time error because the variable is already borrowed by the first use.

Complete Type Inference



```
let gradeDB = Dbm.opendbm "gradeDB" ... in
Dbm.add &!gradeDB "math" 42;
(* ... *)
Dbm.close gradeDB;
(* ... *)
print_int (Dbm.find &gradeDB "literature") (* x compile-time error! *)
```

Table of contents

1. Linearity through kinds
2. Functions and captures
3. Borrows and regions
4. Inference and constraints

Linearity through kinds

Kinds determine usage:

- Linear (**lin**): Used exactly once [1]
- Affine (**aff**): Used at most once [0 – 1]
- Unrestricted (**un**): Used arbitrarily many time [0 – ∞]

Examples:

```
type database : lin  
type string : un
```

Linearity through kinds

Kinds determine usage:

- Linear (**lin**): Used exactly once [1]
- Affine (**aff**): Used at most once [0 – 1]
- Unrestricted (**un**): Used arbitrarily many time [0 – ∞]

Examples:

```
type ('a : 'k) list : 'k
```

Linearity through kinds

Kinds determine usage:

- Linear (**lin**): Used exactly once [1]
- Affine (**aff**): Used at most once [0 – 1]
- Unrestricted (**un**): Used arbitrarily many time [0 – ∞]

Examples:

```
type ('a : 'k) list : 'k  
val create_list : ('a : un) => int -> 'a -> 'a list
```

Linearity through kinds

Kinds determine usage:

- Linear (**lin**): Used exactly once [1]
- Affine (**aff**): Used at most once [0 – 1]
- Unrestricted (**un**): Used arbitrarily many time [0 – ∞]

Examples:

```
type ('a : 'k) list : 'k  
val create_list : ('a : un) => int -> 'a -> 'a list
```

We also use **subkinding**: **un** \leq **aff** \leq **lin**

Table of contents

1. Linearity through kinds
2. Functions and captures
3. Borrows and regions
4. Inference and constraints

Functions and captures

```
let gradeDB = Dbm.open ...
```

```
let log_n_close msg =  
  printf "Closing: %s" msg;  
  Dbm.close gradeDB
```


Functions and captures

```
let gradeDB = Dbm.open ...
```

```
let log_n_close msg =  
  printf "Closing: %s" msg;  
  Dbm.close gradeDB
```

Capture!

Functions and captures

```
let gradeDB = Dbm.open ...
```

```
let log_n_close msg =  
  printf "Closing: %s" msg;  
  Dbm.close gradeDB
```

Capture!

A rounded rectangular box labeled "Capture!" has two arrows. One arrow points from the box to the `gradeDB` variable in the `Dbm.close` call of the `log_n_close` function. The other arrow points from the box to the `gradeDB` variable in the `Dbm.open` call above it.

We infer the type:

```
val log_n_close : string  $\xrightarrow{\text{lin}}$  unit
```

Usage mode

A rounded rectangular box labeled "Usage mode" has an arrow pointing to the $\xrightarrow{\text{lin}}$ annotation in the type signature of `log_n_close`.

Functions and captures

```
let gradeDB = Dbm.open ...
```

```
let log_n_close msg =  
  printf "Closing: %s" msg;  
  Dbm.close gradeDB
```

Capture!

A rounded rectangle labeled "Capture!" has two arrows pointing to the variable `gradeDB` in the function definition above. One arrow points from the top of the box to the `gradeDB` in `Dbm.open ...`. The other arrow points from the right side of the box to the `gradeDB` in `Dbm.close gradeDB`.

We infer the type:

```
val log_n_close : string  $\xrightarrow{\text{lin}}$  unit
```

Warning: Does not say anything about the arguments!!

A red rounded rectangle containing the text "Warning: Does not say anything about the arguments!!" has a red arrow pointing to the `lin` annotation in the type signature above.

Usage mode

A rounded rectangle labeled "Usage mode" has an arrow pointing to the `lin` annotation in the type signature above.

Table of contents

1. Linearity through kinds
2. Functions and captures
3. Borrows and regions
4. Inference and constraints

Borrows

A borrow is a temporary loan of a resource a

- **Shared** borrows `&a` are for observing the resource
- **Exclusive** borrows `&!a` are for modifying the resource

Borrows

A borrow is a temporary loan of a resource a

- **Shared** borrows `&a` are for observing the resource
- **Exclusive** borrows `&!a` are for modifying the resource

A correct usage of borrows:

```
let avg =  
  (Dbm.find &gradeDB "math" + Dbm.find &gradeDB "compsci") / 2  
  (* ✓ Multiple shared borrows *)  
in  
Dbm.add &!gradeDB "average" avg (* ✓ One exclusive borrow *)
```

Unrestricted - **un**

Affine - **aff**

Borrows – Example of uses

Rule 1: Cannot use a borrow and the resource itself simultaneously

```
let gradeDB = ... in  
f (gradeDB, &gradeDB) (* x Conflicting use and borrow! *)
```

Borrows – Example of uses

Rule 2: Cannot use an exclusive borrow and any other borrow simultaneously

```
let gradeDB = ... in  
f (&!gradeDB, &gradeDB) (* x Conflicting borrows! *)
```


Borrows – Example of uses

Rule 3: Borrows must not escape

```
let f () =  
  let gradeDB = ... in  
  let x = (&gradeDB, "mygrades") in  
  x  
  (* x Borrow escaping its scope! *)
```

Borrows – Example of uses

Rule 3: Borrows must not escape

```
let f () =  
  let gradeDB = ... in  
  { | let x = (&gradeDb, "mygrades") in  
    x | }  
  (* x Borrow escaping its scope! *)
```

Borrows – Example of uses

Rule 3: Borrows must not escape

```
let f () =  
  let gradeDB = ... in &gradeDb : &(database, un2)  
  { | let x = (&gradeDb, "mygrades") in  
    x | }  
  (* x Borrow escaping its scope! *) Region nesting level: 1
```

Indexed kinds ensure that borrows do not escape!

Borrows of index 2 cannot escape a region of index 1.

Everything together

The Database API:

```
type database : lin
```

```
val find : &(database, 'k) -> string  $\xrightarrow{'k}$  int
```

```
val add : &!(database, 'k) -> string  $\xrightarrow{'k}$  int  $\xrightarrow{'k}$  unit
```

Everything together

The Database API:

```
type database : lin
val find : &(database, 'k) -> string  $\xrightarrow{'k}$  int
val add : &!(database, 'k) -> string  $\xrightarrow{'k}$  int  $\xrightarrow{'k}$  unit
```

A simple use:

```
let gradeDB = ... in
let avg =
  (Dbm.find &gradeDB "math" + Dbm.find &gradeDB "compsci") / 2

in
Dbm.add &!gradeDB "average" avg
```

Everything together

The Database API:

```
type database : lin
val find : &(database, 'k) -> string  $\xrightarrow{'k}$  int
val add : &!(database, 'k) -> string  $\xrightarrow{'k}$  int  $\xrightarrow{'k}$  unit
```

A simple use:

```
let gradeDB = ... in
let avg =
  let grade subject = Dbm.find &gradeDB subject in (* ✓ Capture *)
  (grade "math" + grade "compsci") / 2
in
Dbm.add &!gradeDB "average" avg
```

Everything together

The Database API:

```
type database : lin
val find : &(database, 'k) -> string  $\xrightarrow{'k}$  int
val add : &!(database, 'k) -> string  $\xrightarrow{'k}$  int  $\xrightarrow{'k}$  unit
```

A simple use:

```
let gradeDB = ... in
let avg =
  let grade = Dbm.find &gradeDB in (* ✓ Partial application *)
  (grade "math" + grade "compsci") / 2
in
Dbm.add &!gradeDB "average" avg
```

Everything together

The Database API:

```
type database : lin
val find : &(database, 'k) -> string  $\xrightarrow{'k}$  int
val add : &!(database, 'k) -> string  $\xrightarrow{'k}$  int  $\xrightarrow{'k}$  unit
```

A simple use:

```
let average db subjects =
  List.map (Dbm.find db) subjects / List.length subjects
let main () =
  let gradeDB = ... in
  let avg = average &gradeDB ["math"; "compsci"; ...] in
  Dbm.add &!gradeDB "average" avg
```


Everything together

The Database API:

```
type database : lin
val find : &(database, 'k) -> string  $\xrightarrow{'k}$  int
val add : &!(database, 'k) -> string  $\xrightarrow{'k}$  int  $\xrightarrow{'k}$  unit
```

A simple use:

```
let average db subjects =
  List.map (Dbm.find db) subjects / List.length subjects
let main () =
  let gradeDB = ... in
  let avg = { | average &gradeDB ["math"; "compsci"; ...] | } in
  { | Dbm.add &!gradeDB "average" avg | }
```

No type annotation

Disjoint regions

Table of contents

1. Linearity through kinds
2. Functions and captures
3. Borrows and regions
4. Inference and constraints

Inference in action

```
let average db subjects =  
  List.map (Dbm.find db) subjects / List.length subjects  
  
let main () =  
  let gradeDB = ... in  
  let avg = average &gradeDB ["math"; "compsci"; ...] in  
  Dbm.add &!gradeDB "average" avg
```

1. Elaborate
regions

```
let average db subjects =  
  List.map (Dbm.find db) subjects / List.length subjects  
  
let main () =  
  let gradeDB = ... in  
  let avg = { | average &gradeDB ["math"; "compsci"; ...] | } in  
  { | Dbm.add &!gradeDB "average" avg | }
```

1. Infer the placement of region based on the position of borrows and the borrowing rules.

Inference in action

```
let average db subjects =  
  List.map (Dbm.find db) subjects / List.length subjects  
  
let main () =  
  let gradeDB = ... in  
  let avg = average &gradeDB ["math"; "compsci"; ...] in  
  Dbm.add &!gradeDB "average" avg
```

1. Elaborate
regions

```
let average db subjects =  
  List.map (Dbm.find db) subjects / List.length subjects  
  
let main () =  
  let gradeDB = ... in  
  let avg = { | average &gradeDB ["math"; "compsci"; ...] | } in  
  { | Dbm.add &!gradeDB "average" avg | }
```

2. Generate
constraints

$$\begin{aligned}\Gamma &= (\alpha_f : \kappa_f)(\alpha_x : \kappa_x) \dots \\ C &= (\alpha_f \leq \gamma \xrightarrow{\kappa_1} \beta) \wedge (\gamma \leq \alpha_x) \\ &\quad \wedge (\beta \times \alpha_x \leq \alpha_r) \wedge (\kappa_x \leq U) \\ &\quad \wedge \dots\end{aligned}$$

2. Generate custom constraints based on HM(X).

Inference in action

```
let average db subjects =  
  List.map (Dbm.find db) subjects / List.length subjects  
  
let main () =  
  let gradeDB = ... in  
  let avg = average &gradeDB ["math"; "compsci"; ...] in  
  Dbm.add &!gradeDB "average" avg
```

1. Elaborate
regions

```
let average db subjects =  
  List.map (Dbm.find db) subjects / List.length subjects  
  
let main () =  
  let gradeDB = ... in  
  let avg = { | average &gradeDB ["math"; "compsci"; ...] | } in  
  { | Dbm.add &!gradeDB "average" avg | }
```

2. Generate
constraints

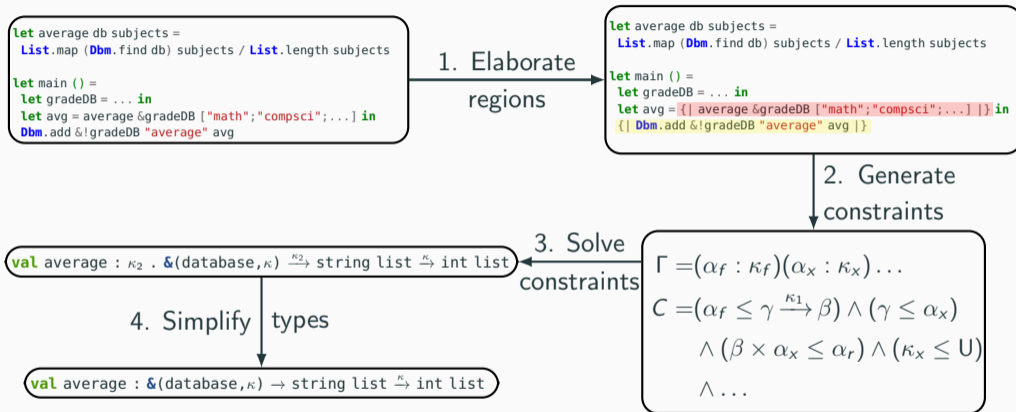
```
val average :  $\kappa_2$  . &(database,  $\kappa$ )  $\xrightarrow{\kappa_2}$  string list  $\xrightarrow{\kappa}$  int list
```

3. Solve
constraints

$$\Gamma = (\alpha_f : \kappa_f)(\alpha_x : \kappa_x) \dots$$
$$C = (\alpha_f \leq \gamma \xrightarrow{\kappa_1} \beta) \wedge (\gamma \leq \alpha_x)$$
$$\wedge (\beta \times \alpha_x \leq \alpha_r) \wedge (\kappa_x \leq U)$$
$$\wedge \dots$$

3. Solve the constraints using a custom algorithm and obtain principal type schemes.

Inference in action



4. Simplify the obtained type scheme by leveraging subkinding

The Affe language – Summary

Prototype: <https://affe.netlify.com/>

- ✓ Linearity, Closures, Borrows and Regions
⇒ Good support for both imperative and functional programming
- ✓ Support managed and unmanaged objects
- ✓ Principal type inference
- ✗ No flow sensitivity
- ✗ No concurrency story (yet)

Many examples in the paper: files, session types, semi-persistent arrays, iterators, connection pools, ...

The theory – Summary

- A Syntax-directed type system for Affe
 - ⇒ How to encode borrows into an ML-style type-system
- A formal semantics that takes allocations into account (+ proof of soundness)
- An inference algorithm for Affe:
 - An extension of $HM(X)$ with kinds
 - A novel constraint systems to encode linearity and borrows
 - A constraint solving algorithm, and its proof of completeness

Close(Talk)