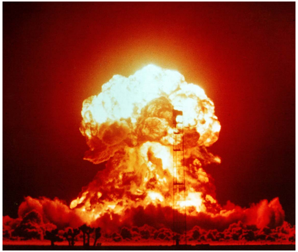
A Substructural Type and Effect System

Orpheas van Rooij[1,2]   Robbert Krebbers[2]

[1]University of Edinburgh

[2]Radboud University

February 2025

References + Effect Handlers = 

Deallocatable References 😬

```
alloc :: a → Ref a
bar :: Ref Int → Bool
free :: Ref a → a
read :: Ref a → a

foo :: Unit → Int
foo () =
    let (x :: Ref Int) ← alloc 0
    in if bar x then free x else read x
```

What can go wrong with this code?

Deallocatable References 😬

```
alloc :: a → Ref a
bar :: Ref Int → Bool
free :: Ref a → a
read :: Ref a → a

foo :: Unit → Int
foo () =
    let (x :: Ref Int) ← alloc 0
    in if bar x then free x else read x
```

What can go wrong with this code?
Many things!

- What if bar frees x?
  - **Double free**
  - **Use after free**

Google Pays Out $55,000 Bug Bounty for Chrome Vulnerability

Google has released a Chrome 133 update to address four high-severity vulnerabilities reported by external researchers.

By Ionut Arghire
February 13, 2025

Google on Wednesday announced the rollout of a Chrome browser update that resolves four high-severity vulnerabilities that were reported by external researchers.

The first issue is a use-after-free bug in the V8 JavaScript engine, tracked as

TRENDING

1 Palo Alto Network Exploitation of Fire Vulnerability

2 New Windows Zer Exploited by Chine Security Firm

3 Finastra Starts Not Impacted by Recer Breach

4 Xerox Versalink Pri Vulnerabilities Ena Movement

5 Microsoft Warns o XCSSET macOS Ma

6 New FinalDraft Ma Spotted in Espiona

7 Russian State Hacl Organizations With Phishing

8 Singulr Launches V Funding for AI Secu Governance Platfo

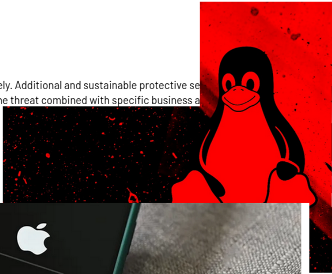
New StackRot Linux kernel flaw allows privilege escalation

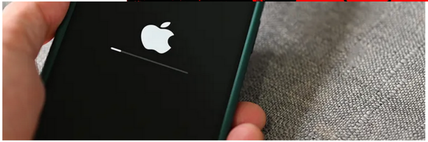
July 6, 2023

New Pre-Auth Double Free Vulnerability on OpenSSH

Threat Level Description

Threat Level: High – An attack is highly likely. Additional and sustainable protective se
measures reflecting the broad nature of the threat combined with specific business a





(Adobe Stock)

Apple on Jan. 27 patched its first zero-day of 2025, a bug that the company confirmed was actively exploited in the wild on iOS devices.

The bug — CVE-2025-24085 — was a "use after free" issue that was addressed with improved memory management. The issue is fixed in visionOS 2.3, iOS 18.3 and iPadOS 18.3, macOS Sequoia 15.3, watchOS 11.3, tvOS 18.3.

CVE-2022-30211: Windows L2TP VPN Memory Leak and Use after Free Vulnerability

Forensics

MalBot Aug 2022

Nettitude discovered a Memory Leak turned Use after Free (UaF) bug in the Microsoft implementation of the L2TP VPN protocol. The vulnerability affects most server and desktop versions of Windows, dating back to Windows Server 2008 and Windows 7 respectively. This could result in a Denial of Service (DoS) condition or could potentially be exploited to achieve Remote Code Execution (RCE).

Mozilla fixes Firefox zero-day actively exploited in attacks

Mozilla has issued an emergency security update for the Firefox browser to address a critical use-after-free vulnerability that is currently exploited in attacks.

BILL TOULAS OCTOBER 09, 2024 01:34 PM 1

SQLite patches use-after-free bug that left apps open to code execution, denial-of-service exploits

Adam Bannister 16 February 2021 at 15:30 UTC
Updated: 18 May 2021 at 13:07 UTC

Vulnerabilities Cyber-attacks Windows

More than one trillion SQLite databases potentially active in myriad operating systems, browsers, and applications

Problem Invalid memory operations can introduce security-critical bugs.

Problem Invalid memory operations can introduce security-critical bugs.

Solution A compiler that statically detects and forbids invalid memory operations.

$\Rightarrow$ Using a (substructural) type system!

Substructural Type Systems

Types and all that

What is a type?

- **Simple Types:** $\texttt{Unit}, \texttt{Int}, \texttt{Int} \to \texttt{Bool}$
  Give us information on the contents of variables/memory cells.
- **Substructural Types:** $\underline{\texttt{Ref Int}}, \underline{\texttt{Int} \multimap \texttt{Int}}$
  Additionally tells us how many times we can use things.
- **Effect Types:** $\texttt{Int} \xrightarrow{\langle \text{choose}:\texttt{Unit}\Rightarrow\texttt{Bool}\rangle} \texttt{Unit}$
  Additionally tells us what kinds of effects it performs.
- **Dependent types**, **Session types**, ...

Type Systems and all that

What is a type system?

⇒ A way to **statically**/gradually/dynamically enforce that types are obeyed.

⇒ Described using formal typing rules that we compose to get typing derivations.

$$\textsc{Abs}\ \frac{\mathtt{x} : \tau, \Gamma \vdash e : \kappa}{\Gamma \vdash \backslash\mathtt{x} \to e : \tau \to \kappa}$$

Type Systems and all that

Simple Types: Ensure content of variables/memory satisfies their types:

Accepted ✓	Rejected ⊗
`True :: Bool`	`256 :: Bool`
`(\ x → x + 1) 0`	`True 0`

Substructural Types: What exactly do we enforce here?

- How to treat `Ref Int`, `Int ⊸ Int`?

Affine Reference Types

To rule out memory errors related to references we need to treat (Ref a) affinely

We must use references at most once.

Accepted ✓	Rejected ⊗
`let (x :: Ref Int) ← alloc 0 in x`	`let (x :: Ref Int) ← alloc 0 in (x,x)`
`let (x :: Ref Int) ← alloc 0 in free x`	`let (x :: Ref Int) ← alloc 0 in free x; free x`

`let x ← alloc 0`
`in (\ () → free x)`
	`let x ← alloc 0 in`
	`let f ← (\ () → free x)`
	`in f (); f ()`

Affine Function Types

We said:

Accepted ✓	Rejected ⊗

`let x ← alloc 0`
`in (\ () → free x)`
	`let x ← alloc 0 in`
	`let f ← (\ () → free x)`
	`in f (); f ()`

Problem What type should we give to `f`?

Affine Function Types

We said:

Accepted ✓	Rejected ⊗

`let x ← alloc 0`
`in (\ () → free x)`
	`let x ← alloc 0 in`
	`let f ← (\ () → free x)`
	`in f (); f ()`

Problem What type should we give to `f`?

Solution Introduce a new affine function type ($\multimap$):

- $\underline{\texttt{a} \multimap \texttt{b}}$: Must be called *at most* once
- $\texttt{a} \to \texttt{b}$: Can be called any number of times

Deallocatable References ☺

```
foo :: Unit → Int
foo () =
    let (x :: Ref Int) ← alloc 0
    in if bar x then free x⊗ else read x⊗
```

x is already used up by bar

Our substructural type system **rejects** this program.

⇒ The responsibility for deallocating x falls to bar.

Algebraic Effects and Handlers

Exception Handlers

Exceptions allow us to raise an error anywhere in the code:

$$\texttt{add} :: \texttt{UInt} \rightarrow \texttt{UInt} \xrightarrow{\langle \texttt{Overflow} \rangle} \texttt{UInt}$$
$$\texttt{add}\ x\ y = \texttt{if}\ (\texttt{UINT\_MAX} - x) < y\ \texttt{then raise Overflow (x,y) else}\ x + y$$

Exception Handlers

Exceptions allow us to raise an error anywhere in the code:

```
add :: UInt → UInt -⟨Overflow⟩→ UInt
add x y = if (UINT_MAX − x) < y then raise Overflow (x,y) else x + y
```

And allow us to install handlers to service the error:

```
complexCode :: Unit → Maybe UInt
complexCode () =
   handle ...add (work1 ()) (work2 ())... by
     Overflow (x,y) → printf "Overflow detected:%d + %d" x y; Nothing
    | ret x → Just x
```

Effect Handlers

Exception handlers cannot resume the expression that raised the exception.
But what if we could?

⇒ When overflow occurs, return `UINT_MAX` as the result of `add`

Effect Handlers

Exception handlers cannot resume the expression that raised the exception.
But what if we could?

⇒ When overflow occurs, return `UINT_MAX` as the result of `add`

```
complexCode :: Unit → Maybe UInt
complexCode () =
    handle ... add (work1 ()) (work2 ()) ... by
      Overflow (x,y) k → k UINT_MAX
     | ret x → Just x
```

Algebraic effects and handlers

It turns out that algebraic effects and handlers can express:

- Non-determinism
- Cooperative Concurrency
- State
- and more . . .

Retrofitted to OCaml and to research languages Links, Koka, Eff

Algebraic Effects

The source of the effects comes from *operations*:

$$\texttt{Choose} :: \texttt{Unit} \Rightarrow \texttt{Bool}$$

The function `chooseInt (m,n)` returns either `m` or `n`:

$$\begin{array}{l}\texttt{chooseInt} :: (\texttt{Int}, \texttt{Int}) \xrightarrow{\langle \texttt{Choose}::\texttt{Unit}\Rightarrow\texttt{Bool}\rangle} \texttt{Int} \\ \texttt{chooseInt (x,y)} = \texttt{if do Choose () then x else y}\end{array}$$

Algebraic Effects and Handlers

Handlers give meaning to operations such as `Choose`:

```
chooseFirst = handle ... by Choose () k → k True
```

`k` is a *one-shot* continuation: represents the remaining unevaluated program.
e.g. in `chooseInt(m,n)`:

```
if do Choose () then m else n
                           k
```

```
handle chooseInt(1,2) by Choose () k → k True
```

Algebraic Effects and Handlers

```
handle chooseInt(1,2) by Choose () k → k True
```

steps to

```
handle (if do Choose () then 1 else 2) by Choose () k → k True
```

Algebraic Effects and Handlers

```
handle chooseInt(1,2) by Choose () k → k True
```

steps to

```
handle (if do Choose () then 1 else 2) by Choose () k → k True
```

steps to

```
(\x → if x then 1 else 2) True
```

```
handle chooseInt(1,2) by Choose () k → k True
```

steps to

```
handle (if do Choose () then 1 else 2) by Choose () k → k True
```

steps to

```
(\ x → if x then 1 else 2) True
```

steps to

```
if True then 1 else 2
```

Algebraic Effects and Handlers

```
handle chooseInt(1,2) by Choose () k → k True
```

steps to

```
handle (if do Choose () then 1 else 2) by Choose () k → k True
```

steps to

```
(\ x → if x then 1 else 2) True
```

steps to

```
if True then 1 else 2
```

steps to

```
1
```

Alternatively we can collect all the possible results from calls to `Choose`:

```
collectAll = handle ... by
               Choose () k → k True ++ k False
             | ret x → [x]
```

`k` is called a *multi-shot* continuation here.

Algebraic Effects and Handlers

```
handle chooseInt(1,2) by
  Choose () k → k True ++ k False
 | ret x → [x]
```

```
handle chooseInt(1,2) by
  Choose () k → k True ++ k False
 | ret x → [x]
```

steps to

```
handle (if do Choose () then 1 else 2) by
  Choose () k → k True ++ k False
 | ret x → [x]
```

Algebraic Effects and Handlers

```
handle chooseInt(1,2) by
  Choose () k → k True ++ k False
 | ret x → [x]
```

steps to

```
handle (if do Choose () then 1 else 2) by
  Choose () k → k True ++ k False
 | ret x → [x]
```

steps to

```
(\x → [if x then 1 else 2]) True ++ (\x → [if x then 1 else 2]) False
```

Algebraic Effects and Handlers

```
handle chooseInt(1,2) by
  Choose () k → k True ++ k False
 | ret x → [x]
```

steps to

```
handle (if do Choose () then 1 else 2) by
  Choose () k → k True ++ k False
 | ret x → [x]
```

steps to

```
(\x → [if x then 1 else 2]) True ++ (\x → [if x then 1 else 2]) False
```

steps to

```
[1,2]
```

Problem Statement

Problem Statement

What happens when we have both (deallocatable) references and effect handlers?

$$\mathtt{bad} :: \underline{\mathtt{Ref\ Int}} \xrightarrow{\langle \mathtt{Choose} :: \mathtt{Unit} \Rightarrow \mathtt{Bool} \rangle} \mathtt{Int}$$
$$\mathtt{bad}\ \mathtt{r} = \mathtt{let}\ \mathtt{x} \leftarrow \mathtt{chooseInt(1,2)}\ \mathtt{in}\ \mathtt{free}\ \mathtt{r} + \mathtt{x}$$

Problem Statement

What happens when we have both (deallocatable) references and effect handlers?

$$\texttt{bad} :: \underline{\texttt{Ref Int}} \xrightarrow{\langle \texttt{Choose} :: \texttt{Unit} \Rightarrow \texttt{Bool} \rangle} \texttt{Int}$$
$$\texttt{bad}\ \texttt{r} = \texttt{let}\ \texttt{x} \leftarrow \texttt{chooseInt(1,2)}\ \texttt{in}\ \texttt{free}\ \texttt{r} + \texttt{x}$$

Depends on the handler we install:

One-shot Choose	Multi-shot Choose
`chooseFirst (bad (alloc 0))` No memory errors occur	`collectAll (bad (alloc 0))` Double free occurs!

Problem Statement

When evaluating `collectAll (bad (alloc 0))` we get a double-free:

$$[\texttt{let x} \leftarrow 1\ \texttt{in free r} + \texttt{x}]_{\texttt{k True}} +\!\!+ [\texttt{let x} \leftarrow 2\ \texttt{in free r} + \texttt{x}]_{\texttt{k False}}$$

Because the reference `r` is *captured in the multi-shot continuation!*

Problem Substructurally treated references and multi-shot effects can still break the memory safety guarantees.

Problem Substructurally treated references and multi-shot effects can still break the memory safety guarantees.

Solution An effect system must also be substructural aware:
Needs to distinguish between *one-shot* and *multi-shot* effects.

Solution: Substructural (Type and Effect) System

Solution

A affine type and effect system that distinguishes *one-shot* from *multi-shot* effects:

One-shot	Multi-shot
$\mathsf{choose} : \texttt{Unit} \multimap \texttt{Bool}$	$\mathsf{choose} : \texttt{Unit} \Rightarrow \texttt{Bool}$

- **One-shot effect:** Can only be handled in a one-shot way (e.g. `chooseFirst`)
- **Multi-shot effect:** Can be handled in a multi-shot way (e.g. `collectAll`)
  - + Forbid references from being captured in multi-shot continuations.

Solution

The `bad` function is **not** typeable:

$$\texttt{bad} :: \underline{\texttt{Ref Int}} \xrightarrow{\langle \texttt{Choose}::\texttt{Unit} \Rightarrow \texttt{Bool} \rangle} \texttt{Int}$$

$$\texttt{bad}\ \texttt{r} = \texttt{let}\ \texttt{x} \leftarrow \texttt{chooseInt}(1,2)\ \texttt{in}\ \texttt{free}\ \texttt{r}_{\otimes} + \texttt{x}$$

`r` is captured in a multi-shot continuation

Instead `Choose` must be typed as one-shot ($\multimap$):

$$\texttt{good} :: \underline{\texttt{Ref Int}} \xrightarrow{\langle \texttt{Choose}::\texttt{Unit} \multimap \texttt{Bool} \rangle} \texttt{Int}$$

$$\texttt{good}\ \texttt{r} = \texttt{let}\ \texttt{x} \leftarrow \texttt{chooseInt}(1,2)\ \texttt{in}\ \texttt{free}\ \texttt{r} + \texttt{x}$$

Semantic Typing

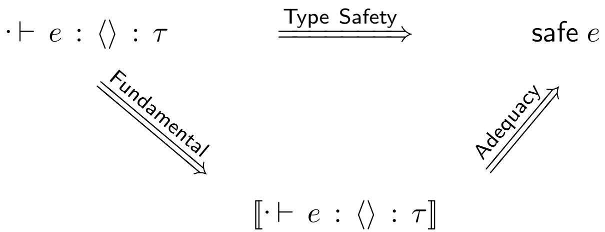
Semantic Typing

$$\vdash e : \langle\rangle : \tau \quad \xRightarrow{\text{Type Safety}} \quad \text{safe } e$$

safe e here should mean:

- $\Rightarrow$ No usual type errors.
- $\Rightarrow$ No effects are left unhandled.
- $\Rightarrow$ **Use-after-free, double-free errors do not happen!**

Semantic Typing



Types, effects, typing judgements are interpreted in the logic of Iris.

Fundamental: Each typing rule becomes a lemma.
Adequacy: Semantic type judgements imply expression safety.

The logic of Iris

We interpret types and judgments in the *Iris* separation logic:

$$
\begin{aligned}
P, Q, R \in \mathsf{iProp} := {} & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid \forall \mathtt{x}.\, P \mid \exists \mathtt{x}.\, P \mid \\
& \ell \mapsto v \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \mathsf{wp}\; e \;_{\Psi}\{\Phi\} \mid \\
& \Box\, P \mid \triangleright\, P \mid \boxed{P}^{\mathcal{N}} \mid \ldots
\end{aligned}
$$

Iris from the ground up

A modular foundation for higher-order concurrent separation logic

RALF JUNG
MPI-SWS, Germany
(*e-mail:* jung@mpi-sws.org)

ALEŠ BIZJAK
Aarhus University, Denmark
(*e-mail:* abizjak@cs.au.dk)

ROBBERT KREBBERS
Delft University of Technology, The Netherlands
(*e-mail:* mail@robbertkrebbers.nl)

LARS BIRKEDAL
Aarhus University, Denmark
(*e-mail:* birkedal@cs.au.dk)

JACQUES-HENRI JOURDAN
MPI-SWS, Germany
(*e-mail:* jjourdan@mpi-sws.org)

DEREK DREYER
MPI-SWS, Germany
(*e-mail:* dreyer@mpi-sws.org)

The logic of Iris

We interpret types and judgments in the *Iris* separation logic:

$$P, Q, R \in \mathsf{iProp} ::= \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid \forall x.\, P \mid \exists x.\, P \mid$$

$$\ell \mapsto v \mid P * Q \mid P -\!\!* Q \mid \mathsf{wp}\ e\ {}_{\Psi}\{\Phi\} \mid$$

$$\Box P \mid \triangleright P \mid \boxed{P}^{\mathcal{N}} \mid \ldots$$

Iris from the ground up

A modular foundation for higher-order concurrent separation logic

RALF JUNG
MPI-SWS, Germany
(*e-mail:* jung@mpi-sws.org)

ALEŠ BIZJAK
Aarhus University, Denmark
(*e-mail:* abizjak@cs.au.dk)

ROBBERT KREBBERS
Delft University of Technology, The Netherlands
(*e-mail:* mail@robbertkrebbers.nl)

LARS BIRKEDAL
Aarhus University, Denmark
(*e-mail:* birkedal@cs.au.dk)

JACQUES-HENRI JOURDAN
MPI-SWS, Germany
(*e-mail:* jjourdan@mpi-sws.org)

DEREK DREYER
MPI-SWS, Germany
(*e-mail:* dreyer@mpi-sws.org)

The logic of Iris

The type of propositions iProp is resource-aware:

$\ell \mapsto v$ Memory location ℓ is allocated with value v	$P * Q$ P and Q reference distinct locations.
$P \mathrel{-\!\!*} Q$ Resource-aware implication	$\text{wp } e \;{}_{\Psi}\{\Phi\}$ - Expression e diverges or terminates - Performs effects according to protocol Ψ - Its resulting value satisfies predicate Φ

A program logic for effect handlers

Proof rules that allow us to reason about effectful programs:[1]

$$\textsc{WP-Val}\quad \frac{\Phi\, v}{\mathsf{wp}\ v\ \{\Phi\}}$$

$$\textsc{WP-If-True}\quad \frac{\triangleright (\mathsf{wp}\ e_1\ {}_{\Psi}\{\Phi\})}{\mathsf{wp}\ (\texttt{if True then}\ e_1\ \texttt{else}\ e_2)\ {}_{\Psi}\{\Phi\}}$$

$$\textsc{WP-Do}\quad \frac{\Psi\ \mathsf{op}\ v\ \Phi}{\mathsf{wp}\ (\texttt{do}\ \mathsf{op}\ v)\ {}_{\Psi}\{\Phi\}}$$

$$\textsc{WP-Load}\quad \frac{\ell \mapsto w}{\mathsf{wp}\ \texttt{read}\ \ell\ \{v.\ v = w * \ell \mapsto w\}}$$

[1] *de Vilhena and Pottier [POPL'21]*

Interpretation into the logic

Define a logical relation:

$$
\begin{aligned}
[\![\tau]\!], [\![\kappa]\!] \in [\![\mathsf{Type}]\!] &:= \mathsf{Value} \to \mathsf{iProp} \\
[\![\mathtt{Bool}]\!] &:= \lambda\, v.\, v = \mathtt{True} \lor v = \mathtt{False} \\
[\![!\tau]\!] &:= \lambda\, v.\, \Box\,([\![\tau]\!]\, v) \\
[\![\tau \xrightarrow{\rho}\!\!\circ\, \kappa]\!] &:= \lambda\, v.\, \forall\, w.\, [\![\tau]\!]\, w \twoheadrightarrow \mathsf{wp}\ (v\ w)\ _{[\![\rho]\!]}\{\lambda\, v.\, [\![\kappa]\!]\, v\}
\end{aligned}
$$

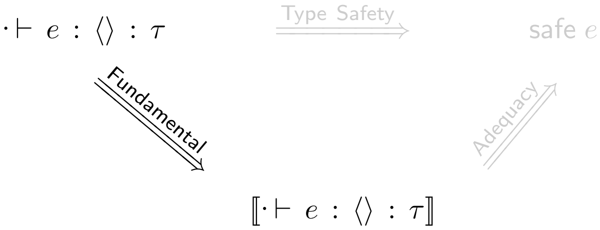
$$
\begin{aligned}
[\![\sigma]\!] \in [\![\mathsf{Effect\,Signature}]\!] &:= \mathsf{Value} \to (\mathsf{Value} \to \mathsf{iProp}) \to \mathsf{iProp} \\
[\![\forall \vec{\mathsf{a}}.\, \tau \Rightarrow \kappa]\!] &:= \lambda\, v\, \Phi.\, \exists\, \vec{\mathsf{a}}.\, [\![\tau]\!]\, v * \Box\,(\forall\, w.\, [\![\kappa]\!]\, w \twoheadrightarrow \Phi\, w) \\
[\![¡\sigma]\!] &:= \lambda\, v\, \Phi.\, \exists\, \Phi'.\, [\![\sigma]\!]\, v\, \Phi' * (\forall\, w.\, \Phi'\, w \twoheadrightarrow \Phi\, w)
\end{aligned}
$$

Typing judgments are interpreted using wp and closing over the open term.

$$
[\![\Gamma \vdash e : \rho : \tau]\!] : \mathit{iProp} \quad := \quad \ldots
$$

The real work

The real work lies in proving the Fundamental lemma:



We have to prove every typing rule:

$$\textsc{Abs}\quad \frac{[\![\mathrm{x} : \tau, \Gamma_1 \vdash e : \rho : \kappa]\!]}{[\![\Gamma_1 \vdash \backslash\,\mathrm{x} \rightarrow e : \langle\rangle : \tau \overset{\rho}{\multimap} \kappa]\!]}$$

Main Takeaways

1. **Deallocatable References** require a *substructural* type system.
2. **Algebraic effects and Handlers** are nice:
   we can implement lots of abstractions using them.
3. **If we want to have both** we need a substructural (type and effect) system!
4. **Semantic Typing** is powerful: can prove type soundness of complicated type systems.

Artifacts Available V1.1 · Artifacts Evaluated Reusable V1.1

Affect: An Affine Type and Effect System

ORPHEAS VAN ROOIJ, Radboud University Nijmegen, Netherlands and University of Edinburgh, UK
ROBBERT KREBBERS, Radboud University Nijmegen, Netherlands