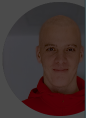
Robust Safety for Move


Marco Patrignani[1] Sam Blackshear[2]


1 UNIVERSITÀ DI TRENTO 2 MystenLabs

Robust Safety for Move



Interested? We're **hiring**!

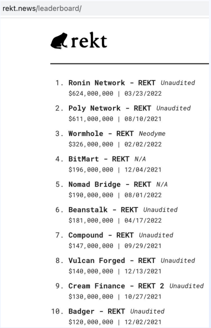
UNIVERSITÀ DI TRENTO

MystenLabs

1 UNIVERSITÀ DI TRENTO 2 MystenLabs

The Move **Language**

Smart contract safety is an existential threat to broader crypto adoption


rekt.news/leaderboard/

rekt

1. **Ronin Network - REKT** *Unaudited*
   $624,000,000 | 03/23/2022
2. **Poly Network - REKT** *Unaudited*
   $611,000,000 | 08/10/2021
3. **Wormhole - REKT** *Neodyme*
   $326,000,000 | 02/02/2022
4. **BitMart - REKT** *N/A*
   $196,000,000 | 12/04/2021
5. **Nomad Bridge - REKT** *N/A*
   $190,000,000 | 08/01/2022
6. **Beanstalk - REKT** *Unaudited*
   $181,000,000 | 04/17/2022
7. **Compound - REKT** *Unaudited*
   $147,000,000 | 09/29/2021
8. **Vulcan Forged - REKT** *Unaudited*
   $140,000,000 | 12/13/2021
9. **Cream Finance - REKT 2** *Unaudited*
   $130,000,000 | 10/27/2021
10. **Badger - REKT** *Unaudited*
    $120,000,000 | 12/02/2021

- 100M+ hacks are routine
- No reason to expect that future smart contract developer will do better...
- Safer SC languages, advanced testing/analysis/verification tools are the only way to grow the dev community in a sustainable way

Smart contracts are unconventional programs

- Smart contracts really only do three things:
  - Define new asset types
  - Read, write, and transfer assets
  - Check access control policies

Thus, need language support for

- Safe abstractions for custom assets, ownership, access control
- Strong isolation–writing safe open-source code that interacts **directly** with code written by motivated attackers

Not common tasks in conventional languages
Not well-supported by existing SC languages

In other smart contract langs, you typically cannot:

- Pass asset as an argument to a function, or return one from a function
- Store an asset in a data structure
- Let a callee function temporarily borrow an asset
- Declare an asset type in contract 1 that is used by contract 2
- Take an asset outside of the contract that created it
    - “trapped” forever in a hash table inside its defining contract

Assets, ownership are the fundamental building blocks of smart contracts, but there’s no vocabulary for describing them!

Move is the first smart contract language to tackle this problem

Assets and ownership encoded via substructural types

"If you **give** me a coin, I will **give** you a car title"

```
fun buy(c: Coin): CarTitle
```

"If you **show** me your title and **pay** a fee, I will **give** you

a car registration"

```
fun register(c: &CarTitle, fee: Coin): CarRegistration { ... }
```

CarTitle, CarRegistration, Coin are user-defined types declared in different modules.

Can flow across trust boundaries without losing integrity

Type system prevents misuse of asset values

Protection against:

Duplication

```
fun f(c: Coin) {
  let x = copy c; // error

  let y = &c;
  let copied = *y; // error
}
```

“Double-spending”

```
fun h(c: Coin) {
  pay(move c);
  pay(move c); // error
}
```

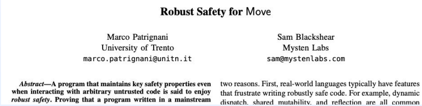
Destruction

```
fun g(c: Coin) {
  c = ... ; // error
  return // error--must move c!
}
```

Ensures that digital assets behave like physical ones

Move design optimizes for safety + predictability

- No dynamic dispatch (no re-entrancy)
- No mixing of aliasing and mutability (like Rust)
- Type/memory/resource safety enforced by bytecode verifier
- Strong isolation aka “robust safety” by default
  - See upcoming CSF ‘23 paper



Robust Safety for Move

Marco Patrignani
University of Trento
marco.patrignani@unitn.it

Sam Blackshear
Mysten Labs
sam@mystenlabs.com

***Abstract*—A program that maintains key safety properties even when interacting with arbitrary untrusted code is said to enjoy *robust safety*. Proving that a program written in a mainstream**

two reasons. First, real-world languages typically have features that frustrate writing robustly safe code. For example, dynamic

- Mathematically ill-defined ops (e.g., int overflow) abort: “SafeMath by default”
- Co-developed with the **Move Prover** formal verification tool (see CAV’20, TACAS ‘21 papers)

Contributions of this Work

Contributions

- formalise Robust Safety (RS) for Move
    - identify the prerequisites for RS

Contributions

- formalise Robust Safety (RS) for Move
  - identify the prerequisites for RS
- prove all Move programs attain RS

Contributions

- formalise Robust Safety (RS) for Move
    - identify the prerequisites for RS
- prove all Move programs attain RS
- implement and evaluate missing tool(s) for RS prerequisites

Contributions

next

- formalise Robust Safety (RS) for Move
  - identify the prerequisites for RS
- prove all Move programs attain RS
- implement and evaluate missing tool(s) for RS prerequisites

Contributions

next

- formalise Robust Safety (RS) for Move
  - identify the prerequisites for RS
- prove all Move programs attain RS

then

- implement and evaluate missing tool(s) for RS prerequisites

Contributions

- formalise Robust Safety (RS) for Move
  - identify the prerequisites for RS

next

- prove all Move programs attain RS

paper

- implement and evaluate missing tool(s) for RS prerequisites

then

Robust Safety (for Move)

What is Robust Safety?

Robust Safety:

maintaining key safety properties even when interacting with arbitrary untrusted code

Bengtson et al. TOPLAS'11, Gordon& Jeffrey JCS'03, Swasey et al. OOPSLA'17 and many more

What is Robust Safety?

Robust Safety:

maintaining key safety properties even when interacting with arbitrary un-trusted code

Bengtson et al. TOPLAS'11, Gordon& Jeffrey JCS'03, Swasey et al. OOPSLA'17 and many more

- key safety properties: programmer-inserted invariants

What is Robust Safety?

Robust Safety:

maintaining key safety properties even when interacting with arbitrary untrusted code

Bengtson et al. TOPLAS'11, Gordon& Jeffrey JCS'03, Swasey et al. OOPSLA'17 and many more

- key safety properties: programmer-inserted invariants
- arbitrary untrusted code: active attacker (with code-like capabilities)

A *(massaged!)* Move **Example**

```
module NextCoin {
  struct Coin has key { value: u64 }
  struct Info has key { tot_supply: u64 }

  spec { ∀c: Coin, info.tot_supply = sum(c.value) }

  public fun mint(... , value: u64): Coin {
    let info = borrow_global_mut< Info> (...);
    info.tot_supply = info.tot_supply + value;
    Coin { value } // invariant broken and restored
  }

  public fun value_mut(coin: &mut Coin): &mut u64 {
    &mut coin.value // not robustly safe!
  }
}
```

Threat Model

- trusted code: the code with invariants (`NextCoin`)

Threat Model

- trusted code: the code with invariants (`NextCoin`)
- attackers: active, write code (e.g., other smart contracts) and interact with the trusted code to break safety

Threat Model

- trusted code: the code with invariants (`NextCoin`)
- attackers: active, write code (e.g., other smart contracts) and interact with the trusted code to break safety
- safety: specified by the programmer-inserted invariants (`spec`)

Local Invariant Verification

- `spec` holds for module `NextCoin` locally

Local Invariant Verification

- `spec` holds for module `NextCoin` locally
  verification done by
  - Move bytecode verifier — Blackshear *et al.* Whitepaper'19
  - Move Prover — Zhong *et al.* CAV'20

Local Invariant Verification

- `spec` holds for module `NextCoin` locally
  verification done by
  - Move bytecode verifier — Blackshear *et al.* Whitepaper'19
  - Move Prover — Zhong *et al.* CAV'20
- (when attackers are not considered)

```
spec { ∀c: Coin, info.tot_supply = sum(c.value) }

public fun mint(... , value: u64): Coin {
  let info = borrow_global_mut< Info> (...);
  info.tot_supply = info.tot_supply + value;
  Coin { value } // invariant broken and restored
}
```

Global Invariant Verification

```
spec { ∀c: Coin, info.tot_supply = sum(c.value) }

public fun value_mut(coin: &mut Coin): &mut u64 {
  &mut coin.value // not robustly safe!
}
```

- `spec` does not hold globally (when attackers are considered)

Global Invariant Verification

```
spec { ∀c: Coin, info.tot_supply = sum(c.value) }

public fun value_mut(coin: &mut Coin): &mut u64 {
  &mut coin.value // not robustly safe!
}
```

- spec does not hold globally
  (when attackers are considered)

```
fun attacker(c: &mut Coin) {
  let value_ref = Coin::value_mut(c);
  *value_ref = *value_ref + 1000; // violates spec!
}
```

From Local to Global Verification

- **Problem**: `value_mut` leaks an invariant-based value

From Local to Global Verification

- **Problem**: `value_mut` leaks an invariant-based value
- **Solution**: enforce encapsulation on invariant-based values

From Local to Global Verification

- **Problem**: `value_mut` leaks an invariant-based value
- **Solution**: enforce encapsulation on invariant-based values

- Trivial? perhaps
- Not-so-trivial? formalising the sufficient conditions for RS and designing an efficient analysis that checks these conditions

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed
- Ω has verified ι locally

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed
- Ω has verified ι locally
- Ω has encapsulated ι

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed
- Ω has verified ι locally
- Ω has encapsulated ι
- for all attackers A running Ω and A respects ι

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed $\vdash \Omega : wt$
- Ω has verified ι locally
- Ω has encapsulated ι
- for all attackers A running Ω and A respects ι

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed $\vdash \Omega : wt$
- Ω has verified ι locally $\Lambda \vdash_{loc} \Omega : \iota$
- Ω has encapsulated ι
- for all attackers A
  running Ω and A
  respects ι

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed $\vdash \Omega : wt$
- Ω has verified ι locally $\Lambda \vdash_{loc} \Omega : \iota$
- Ω has encapsulated ι $\Xi \vdash_{enc} \Omega : \iota$
- for all attackers A
  running Ω and A
  respects ι

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed $\vdash \Omega : wt$
- Ω has verified ι locally $\Lambda \vdash_{loc} \Omega : \iota$
- Ω has encapsulated ι $\Xi \vdash_{enc} \Omega : \iota$
- for all attackers A running Ω and A respects ι $\forall A. \Omega \vdash A : atk$

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed $\vdash \Omega : wt$
- Ω has verified ι locally $\Lambda \vdash_{loc} \Omega : \iota$
- Ω has encapsulated ι $\Xi \vdash_{enc} \Omega : \iota$
- for all attackers A running Ω and A respects ι $\forall A. \Omega \vdash A : atk$ $(\Omega + A) \rightsquigarrow \overline{\alpha}$

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed $\vdash \Omega : wt$
- Ω has verified ι locally $\Lambda \vdash_{loc} \Omega : \iota$
- Ω has encapsulated ι $\Xi \vdash_{enc} \Omega : \iota$
- for all attackers A $\forall A. \Omega \vdash A : atk$
  running Ω and A $(\Omega + A) \rightsquigarrow \overline{\alpha}$
  respects ι $\overline{\alpha} \Vdash \iota$

Robust Safety Definition

A Move module Ω with invariants ι has RS iff:

- Ω is well-typed $\qquad \vdash \Omega : wt$
- Ω has verified ι locally $\qquad \Lambda \vdash_{loc} \Omega : \iota$
- Ω has encapsulated ι $\qquad \Xi \vdash_{enc} \Omega : \iota$
- for all attackers A $\qquad \forall A. \Omega \vdash A : atk$
  running Ω and A $\qquad (\Omega + A) \rightsquigarrow \overline{\alpha}$
  respects ι $\qquad \overline{\alpha} \Vdash \iota$

what are Λ and Ξ?

Tools for Robust Safety in Move

Move **Bytecode Verifier**

Only who declares `Coin` can:

- Create a value of type `Coin`
- "Unpack" a `Coin` into its field(s)
- Acquire a reference to a field of `Coin` via a Rust-style mutable or immutable borrow

Move **Prover for Local Invariants**

- assume global invariants specified by the programmer hold at the entry of each public function
- ensure that they continue to hold at the exit

Move **Prover for Local Invariants**

- assume global invariants specified by the programmer hold at the entry of each public function
- ensure that they continue to hold at the exit

```
spec { ∀c: Coin, info.tot_supply = sum(c.value) }

public fun mint(... , value: u64): Coin {
  Coin { value } // invariant broken
}
```

Encapsulator(s) for …Encapsulation Ξ

- Two classes of attackers:
  - Blockchain-based (imm) Ξ_{imm}
  - non Blockchain-based (mut) Ξ_{mut}

Encapsulator(s) for … Encapsulation Ξ

- Two classes of attackers:
  - Blockchain-based (imm) Ξ_{imm}
  - non Blockchain-based (mut) Ξ_{mut}
- encapsulation:
  when control goes to the attacker

Encapsulator(s) for … Encapsulation Ξ

- Two classes of attackers:
  - Blockchain-based (imm) Ξ_{imm}
  - non Blockchain-based (mut) Ξ_{mut}
- encapsulation:

  when control goes to the attacker

  any resource with an invariant

Encapsulator(s) for ... Encapsulation Ξ

- Two classes of attackers:
  - Blockchain-based (imm) Ξ_{imm}
  - non Blockchain-based (mut) Ξ_{mut}
- encapsulation:

  when control goes to the attacker

  any resource with an invariant

  is not accessible to the attacker

Encapsulator(s) for …Encapsulation

Ξ

- Two classes of attackers:
  - Blockchain-based (imm) Ξ_{imm}
  - non Blockchain-based (mut) Ξ_{mut}
- encapsulation:

  when control goes to the attacker

  calls (mut) and returns (imm & mut)

  any resource with an invariant

  is not accessible to the attacker

Encapsulator(s) for … Encapsulation

Ξ

- Two classes of attackers:
  - Blockchain-based (imm) Ξ_{imm}
  - non Blockchain-based (mut) Ξ_{mut}
- encapsulation:

  when control goes to the attacker

  calls (mut) and returns (imm & mut)

  any resource with an invariant

  using abstract values $\hat{v}$

  is not accessible to the attacker

Encapsulator(s) for …Encapsulation Ξ

- Two classes of attackers:
  - Blockchain-based (imm) Ξ_{imm}
  - non Blockchain-based (mut) Ξ_{mut}
- encapsulation:

  when control goes to the attacker

  calls (mut) and returns (imm & mut)

  any resource with an invariant

  using abstract values $\hat{v}$

  is not accessible to the attacker

  any relevant $\hat{v}$ is not in A's state

Encapsulator Details

- static intraprocedural escape analysis
- abstract values $\hat{v} \in \{\mathsf{NonRef}, \mathsf{OkRef}, \mathsf{InvRef}\}$
  - $\mathsf{NonRef} \sqsubseteq \mathsf{InvRef} \qquad \mathsf{OkRef} \sqsubseteq \mathsf{InvRef}$

Encapsulator Details

- static intraprocedural escape analysis
- abstract values $\hat{v} \in \{\mathsf{NonRef}, \mathsf{OkRef}, \mathsf{InvRef}\}$
    - $\mathsf{NonRef} \sqsubseteq \mathsf{InvRef}$ $\qquad$ $\mathsf{OkRef} \sqsubseteq \mathsf{InvRef}$

(Ξ_{imm}-BorrowFld-Relevant)

$$\frac{f \in \iota}{\Omega, P, \iota, \mathbf{BorrowFld}\,\langle f \rangle \vdash \left\langle \hat{L}, \hat{v}{::}\hat{S} \right\rangle \rightsquigarrow \left\langle \hat{L}, \mathsf{InvRef}{::}\hat{S} \right\rangle}$$

Encapsulator Details

- static intraprocedural escape analysis
- abstract values $\hat{v} \in \{\textsf{NonRef}, \textsf{OkRef}, \textsf{InvRef}\}$
  - NonRef $\sqsubseteq$ InvRef   OkRef $\sqsubseteq$ InvRef

(Ξ_{imm}-BorrowFld-Relevant)

$$\frac{f \in \iota}{\Omega, P, \iota, \mathbf{BorrowFld}\langle f \rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S} \right\rangle \rightsquigarrow \left\langle \hat{L}, \textsf{InvRef}::\hat{S} \right\rangle}$$

(Ξ_{imm}-BorrowFld-Irrelevant)

$$\frac{f \notin \iota}{\Omega, P, \iota, \mathbf{BorrowFld}\langle f \rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S} \right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v}::\hat{S} \right\rangle}$$

Encapsulator Details

- static intraprocedural escape analysis
- abstract values $\hat{v} \in \{\text{NonRef}, \text{OkRef}, \text{InvRef}\}$
  - NonRef $\sqsubseteq$ InvRef   OkRef $\sqsubseteq$ InvRef

(Ξ_{imm}-BorrowFld-Relevant)

$$\frac{f \in \iota}{\Omega, P, \iota, \mathbf{BorrowFld}\langle f \rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S} \right\rangle \rightsquigarrow \left\langle \hat{L}, \text{InvRef}::\hat{S} \right\rangle}$$

(Ξ_{imm}-BorrowFld-Irrelevant)

$$\frac{f \notin \iota}{\Omega, P, \iota, \mathbf{BorrowFld}\langle f \rangle \vdash \left\langle \hat{L}, \hat{v}::\hat{S} \right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v}::\hat{S} \right\rangle}$$

(Ξ_{imm}-Return)

$$\frac{\|\Omega(P).\texttt{rety}\| = n \qquad \forall i \in 1..n.\, \hat{v}_i \neq \text{InvRef}}{\Omega, P, \iota, \mathbf{Ret} \vdash \left\langle \hat{L}, \hat{v_1} :: \hat{v_n}::\hat{S} \right\rangle \rightsquigarrow \left\langle \hat{L}, \hat{v_1} :: \hat{v_n}::\hat{S} \right\rangle}$$

Encapsulator Evaluation

Bench	Mod	Fun	Rec	Instr	Err	T_p	T_e
starcoin	60	431	88	8243	2	3178	10
diem	13	102	19	1830	0	1651	1
mai	45	411	77	7881	0	4209	12
bridge	36	352	85	8060	0	2428	8
blackhole	36	324	72	6030	0	2289	7
alma	35	333	67	6318	0	2102	8
starswap	33	335	67	6617	0	14993	7
meteor	32	323	69	5981	0	1641	7
taohe	11	40	7	305	0	1022	1
stdlib	9	66	5	933	1	1151	1
Total	310	2717	556	52198	3	34664	62

Encapsulator Evaluation

Bench	Mod	Fun	Rec	Instr	Err	T_p	T_e
starcoin	60	431	88	8243	2	3178	10
diem	13	102	19	1830	0	1651	1
mai	45	411	77	7881	0	4209	12
bridge	36	352	85	8060	0	2428	8
blackhole	36	324	72	6030	0	2289	7
alma	35	333	67	6318	0	2102	8
starswap	33	335	67	6617	0	14993	7
meteor	32	323	69	5981	0	1641	7
taohe	11	40	7	305	0	1022	1
stdlib	9	66	5	933	1	1151	1
Total	310	2717	556	52198	3	34664	62

Questions?