The Move Programming Language

Sam Blackshear
Co-founder and CTO of Mysten Labs, creator of Move

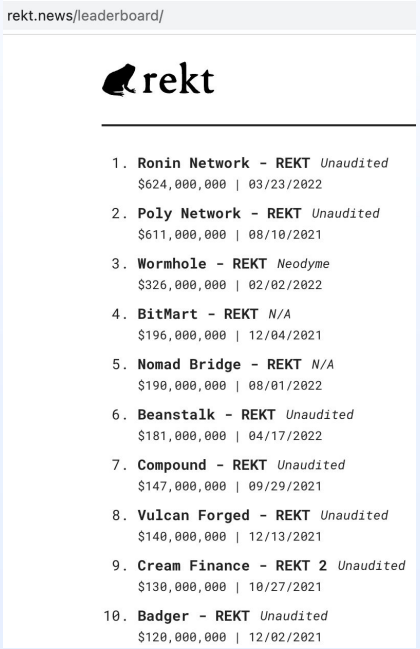
Joint work with collaborators at Mysten, Aptos, Meta, Stanford, Waterloo, StarCoin, 0L

Agenda

- Why Move?
- Move Programming Primer
- Example in Sui Move: flash loan
- Move bytecode, VM ,verifier

Why Move?

Smart contract safety is an existential threat to broader crypto adoption

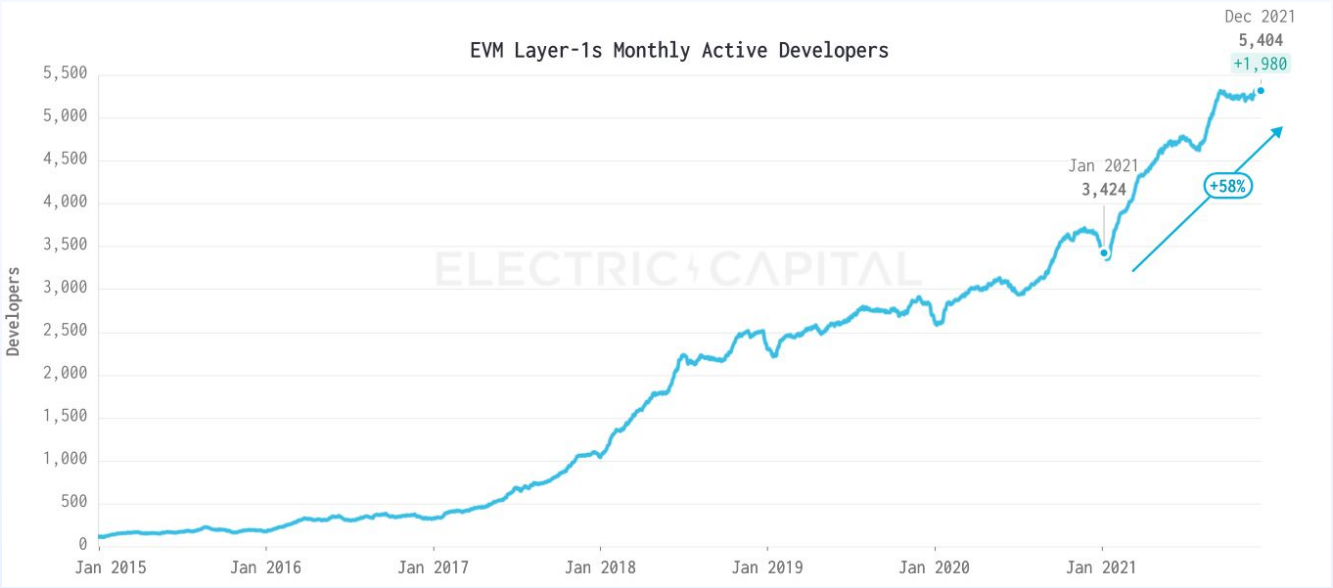

rekt.news/leaderboard/

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

Solidity/EVM network effects over-emphasized



- Small (~5k) dev community [1] relative to conventional langs (e.g., ~16M JS devs)
- Not too late to establish safer, cross-platform industry standard

---

sui

[1] https://medium.com/electric-capital/electric-capital-developer-report-2021-f37874efea6d

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
}
```

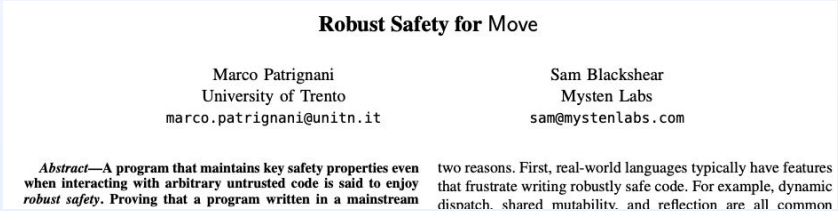
Destruction

```
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 dispatch, shared mutability, and reflection are all common

- 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)

Move is platform-agnostic

- Existing smart contract languages: ~1 language per blockchain
- Languages hardcode account/transaction format, serialization format, cryptography, consensus, ...
  - Use language in new platform = inherit decisions, limitations of the original one
- By contrast, Move is agnostic to all of the above
  - Platform designers can experiment with different choices for the above
  - Programmers can reuse expertise, tooling, libraries across platforms
- Used in 5 blockchains that are very different under the hood:
  - Sui
  - Aptos
  - 0L
  - StarCoin
  - ... and lots of interest from existing + emerging chains (e.g. Solana)

Vision: **Move is the WASM of web3**

Move Programming Primer

Move Code organization

```
module coin::coin {
   …
}
```

```
module defi::flash_lender {
   use coin::Coin;
   …
}
```

```
module defi::escrow {
   use coin::Coin;
   …
}
```

- A *module* is the minimal unit of code abstraction
- Modules declare struct types, functions, constants
- Code/type re-use via imports from other modules
- Multiple modules bundled into single *package* for publish

Move data types

- Primitive types:
  - **bool**
  - unsigned integers: **`u8`**, **`u16`**, **`u32`**, **`u64`**, **`u128`**, **`u256`**
  - asset owner/tx sender: **`address`**
  - Strings: **std::string::String** (UTF8), **std::ascii::String** (ASCII)
- Collection types: **`vector<T>`**, **`Table<K,V>`**
- User defined structs, for example:

```
struct MyStruct {
  int_field: u64,
  struct_field: AnotherStruct
}
```

```
struct AnotherStruct {
  bool_field: bool
}
```

References and Ownership

- Rust-like ownership semantics
  - **&T** - can read **T**
  - **&mut T** - can read or write **T**
  - **T** - can read, write, or *move* (e.g., transfer) **T**
- Structs and data structures cannot contain references
  - No lifetimes, no reborrowing (unlike Rust)
  - Much simpler borrow checker as a result
- No references to references (i.e., **&&T**)

Abilities allow fine-grained struct customization

```
// C can be duplicated. w/o copy, C must be created via pack
struct C has copy { … }

// D can be discarded. w/o drop, D must be eliminated via unpack
struct D has drop { … }

// K can appear in global storage
struct K has key { … }

// S can appear in a field of a key or store type
struct S has store { … }

// "hot potato" must be eliminated in the tx that created it
struct H { … }
```

Generics constrain struct and function defs

```
// Type parameters can have ability constraints
struct Cup<T: store> { t: T }

// Can perform ability-specific operations on unknown types
fun copy<T: copy>(t: &T): T {
  *t
}

// Phantom type parameters are not used in fields
struct Coin<phantom T> { value: u64 }
```

Move function visibility

```
// can be called from a tx and by other modules
public entry fun i()

// can only be called from a tx or within the current module
entry fun j()

// private—only callable within the current module
fun f()

// can be called by other modules
public fun g()

// can be called by other "friend" modules in the same package
public(friend) fun h()
```

Other encapsulation features

```
module m1 {
  struct S { f: u64 }
}

module m2 {
  use m1::S;
  fun bad(s: S) {
    let s2 = S { f: u64 }; // error:can't pack outside m1
    let S { f: u64 } = s; // error:can't unpack outside m1
    let y = &s.f; // error: can't access fields outside m1
  }
}
```

Example in Sui Move: flash loan

Objects are special Move structs

- Mandatory **`key`** *ability* to indicate that this struct is a Sui object
- Mandatory **`id`** field to store globally unique object identifier
- Sui global storage is a **Map<UID, object_bytes>**

```
struct Token has key {
  id: sui::object::UID,
  balance: u64
}
```

Put objects in global storage via transfer

```
fun address_owned_obj<T: key>(t: T, addr: address) {
  transfer::transfer(t, addr)
  // now, t can only be used in tx sent by addr
}

fun shared_owner<T: key>(t: T) {
  transfer::share_object(t)
  // now, t can be used in tx sent by any address
}
```

Defining a flash loan

```
struct FlashLender<phantom T> has key {
  id: UID,
  to_lend: Coin<T>,
  fee: u64
}

struct Receipt {
  lender_id: ID,
  repay_amount: u64
}

struct AdminCap has key {
  id: UID,
  lender_id: ID
}
```

shared object

"Hot potato", value cannot be:
- stored
- dropped
- copied
- transferred

Owned object granting permission to deposit/withdraw from FlashLender

Flash lender creation

```
public fun new<T>(
  to_lend: Coin<T>, fee: u64, ctx: &mut TxContext
) {
  let id = object::new(ctx);
  transfer::transfer(
    AdminCap {
      id: object::new(ctx),
      lender_id: object::uid_to_inner(&id)
    },
    tx_context::sender(ctx)
  );
  transfer::share_object(FlashLender { id, to_lend, fee });
}
```

Requesting a loan

```
public fun loan<T>(
  self: &mut FlashLender<T>,
  amount: u64,
  ctx: &mut TxContext
):(Coin<T>, Receipt<T>) {
  let to_lend = &mut self.to_lend;
  assert!(coin::value(to_lend) >= amount, ELoanTooBig);
  let loan = coin::take(to_lend, amount, ctx);
  let repay_amount = amount + self.fee;
  let receipt = Receipt {
    lender_id: object::id(self),
    repay_amount
  };
  (loan, receipt)
}
```

Repaying a loan

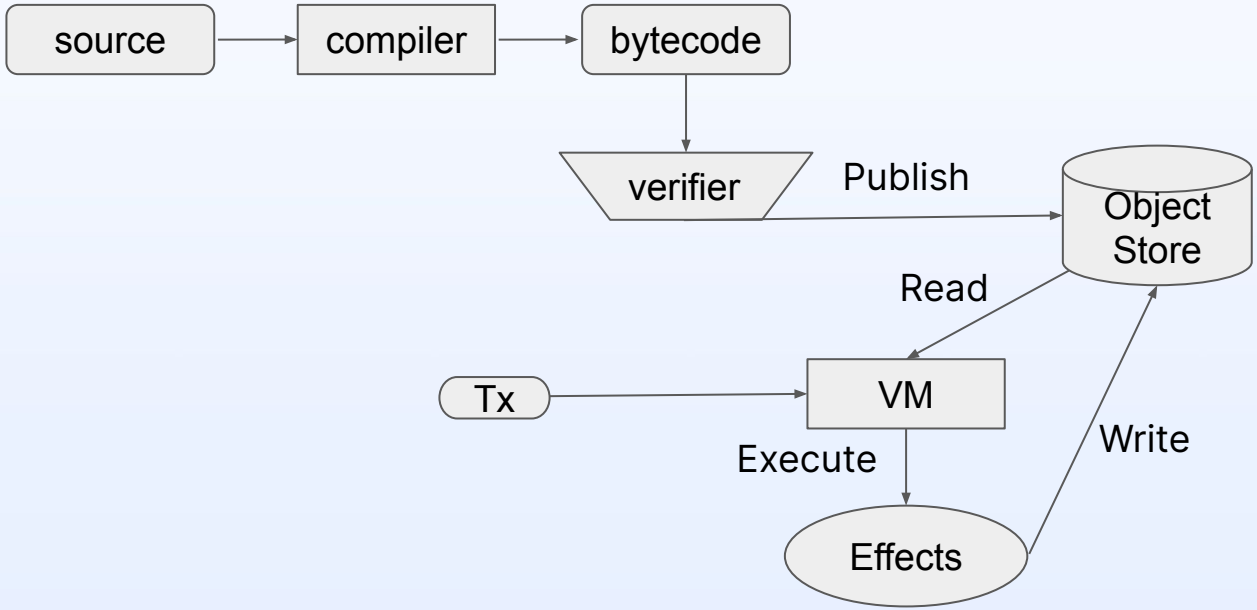
```
public fun repay<T>(
  self: &mut FlashLender<T>,
  payment: Coin<T>,
  receipt: Receipt<T>
) {
  let Receipt { lender_id, repay_amount } = receipt;
  assert!(object::id(self) == lender_id, EWrongLender);
  assert!(coin::value(&payment) == repay_amount, EBadPayment);
  coin::put(&mut self.to_lend, payment)
}
```

Withdrawing profits

```
public fun withdraw<T>(
  self: &mut FlashLender<T>,
  admin_cap: &AdminCap,
  amount: Coin<T>,
  ctx: &mut TxContext
): Coin<T> {
  assert!(object::id(self) == admin_cap.lender_id, EAdminOnly);
  let to_lend = &mut self.to_lend
  assert!(coin::value(&to_lend) >= amount, EWithdrawTooLarge);
  coin::take(to_lend, payment, ctx)
}
```

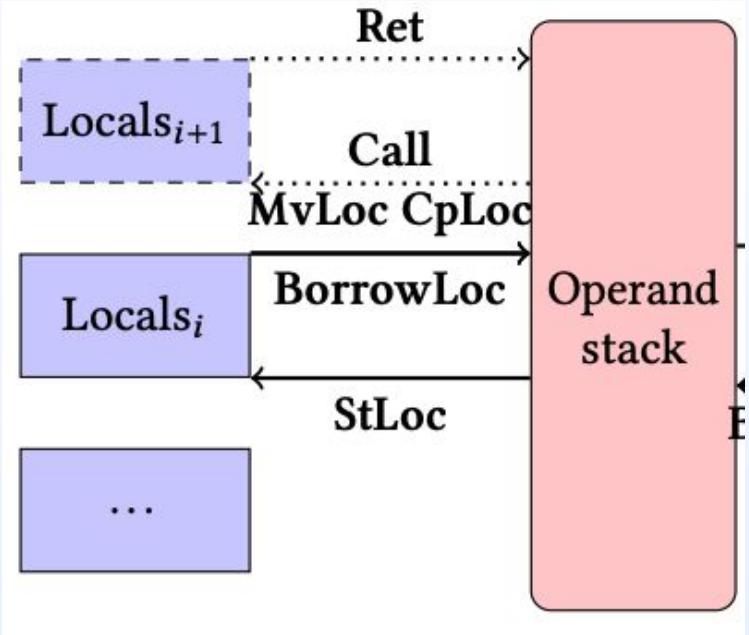
Move bytecode, VM, verifier

Move compile/publish/run toolchain



Move virtual machine

- Stack machine w/ unstructured control-flow
- Each call stack has its own local variables
- Interesting instructions compute w/ data on the operand stack
- All other instructions:
  - move data between locals and operand stack
  - create/destroy call stack frames



Move bytecode instructions

- Locals
  - **MoveLoc<x>**, **CopyLoc<x>**, **StoreLoc<x>**, **BorrowLoc<x>**
- References:
  - **ReadRef**, **WriteRef**, **FreezeRef**
- Structs:
  - **Pack<T>**, **Unpack<T>**, **BorrowField<f>**, **BorrowFieldMut<f>**, **Transfer<T>**
- Control-flow:
  - **Call<p>**, **Ret**, **Br**, **BrTrue**, **BrFalse**, **Abort**
- Stack
  - **Pop**, **Not**, **Add**, **Sub**, **Mul**, **Div**, **BitOr**, **BitAnd**, **Xor**, **Lt**, **Gt**, **Le**, **Ge**, **Or**, **And**, **Eq**, **Neq**, **Shl**, **Shr**

Move bytecode verifier

- Similar idea to JVM, CLR, proof-carrying code
- Type safety
- Ability safety
  - E.g., only types with **copy** ability can use CopyLoc, ReadRef
- Reference safety
  - No dangling references, no memory leaks, referential transparency
- Reducible control-flow graph
- Locals safety:
  - All accessed locals are initialized + not moved
- Stack balancing
  - Callee cannot touch caller’s stack

Move linker

- Checks that module imports match declarations
- All imported functions exist + have the expected signature
- All imported types exist + have the expected abilities
- No cyclic dependencies are created

Key invariants from verifier/VM design

- References can point to locals, but not into the operand stack
  - Operand stack “owns” all of its values
- No persistent references–each reference lives/dies in single tx
- All values are tree-shaped:
  - Roots are structs or primitive values
  - Internal nodes are structs, fields are edges, values are leaves
- Dependency graph is a DAG
- No fallback functions, no re-entrancy, no dynamic dispatch
  - Call targets always exist + are statically known

A word of caution re: Move w/o verifier = ?

- Move gives two kinds of protections:
  - Protecting the programmer from themself
  - Protecting the programmer from other programmers
- (1) is conventional via compiler, static analyzer, tests, …
- (2) is the unique value prop of Move: **all** code is subject to the bytecode verifier, verifier protections are strong
- This allows programmers to safely pass sensitive types across trust boundaries w/o risking loss of integrity
  - E.g., I can pass you a **Coin** and be sure you can't copy it
  - I can pass you a **Receipt** in a flash loan and you'll repay it
- (2) is very hard to achieve if you must interop with existing SC lang

Conclusion

- Move provides safe, convenient abstractions for programming with assets
- Move is platform agnostic and used by four blockchains, more coming in the future
- Move is a great playground for program analysis research without the typical barriers:
  - Mixing aliasing and mutability
  - Dynamic dispatch
  - Reflection
  - Concurrency
- Move community needs more contributors, auditors, researchers, static analyzers, verifiers, fuzzers, testing tools, ...

https://github.com/move-language/move

https://github.com/MystenLabs/awesome-move

https://move-book.com/

https://move-language.github.io/move/