Defense-in-Depth Runtime Safety in Move

Victor Gao, Wolfgang Grieskamp*, Vineeth Kashyap*, George Mitenkov, Teng Zhang, Runtian Zhou, Andrea Cappa, and Marco Ilardi

Aptos Labs, Palo Alto, USA

Abstract. Move is a smart-contract language used to execute transactions on the Aptos blockchain. Move programs execute in a sandboxed VM as typed bytecode. The VM statically verifies foundational safety properties like type safety and reference safety at code loading time. In principle, this design gives strong guarantees for Move. However, the static verification logic is complex and continually evolving with the language; like any software, it is not immune to bugs. In a live blockchain setting, a missed rule violation can translate directly into loss of assets, forged authority, or unrecoverable corruption of on-chain state. For this reason, Aptos relies on defense-in-depth *runtime safety checks* that independently verify the critical invariants during execution, providing protection against latent verifier bugs and malicious bytecode. This paper motivates and describes the runtime safety checks for Move on Aptos.

Keywords: Move · Smart contracts · Type safety · Reference safety.

1 Introduction

The Move on Aptos language [3] was originally developed for the Libra/Diem blockchain at Meta [1, 7, 8] and is now deployed at scale on Aptos [5], a high-throughput Layer-1 blockchain that settles transactions for billions of dollars in user assets on chain. A defining trait of the language is its resource-oriented type discipline: digital assets are represented as Move *resources* whose ownership and movement are tracked by linear-style abilities (`copy`, `drop`, `store`, `key`), ruling out accidental duplication or destruction at compile time, and supported by a regime of mutable and immutable references similar to the Rust language. The Aptos platform builds on this foundation with a system of objects, parallel execution, and a rich on-chain framework written in Move itself [2, 23].

Move has been designed for *offline formal verification*: the Move Prover translates Move programs and their specifications, written in the Move Specification Language, into verification conditions discharged by SMT solvers [12, 14]. The Move Prover can verify rich properties about the smart contracts themselves, but it is inherently offline, applied prior to deployment of the contracts.

Move programs are compiled into typed bytecode executed by the Move VM. The VM performs *static bytecode verification* when a module is loaded, enforcing

* Corresponding authors: Wolfgang Grieskamp (`wg@aptoslabs.com`) and Vineeth Kashyap (`vineeth@aptoslabs.com`).

well-formedness, typing, and Move's reference (§2) discipline. The reference safety verifier [9] is particularly intricate: it uses abstract interpretation to maintain a borrow graph that conservatively approximates the aliasing relationships entailed by control- and data-flow. Keeping these analyses correct is delicate, especially as the language evolves: since the public launch of Aptos, Move has gained enums, closures, and public structs, all of which interact non-trivially with the static verifier. Given the high stakes, Aptos runs a bug bounty program, which has surfaced verifier bugs in the past.

In a permissionless blockchain setting, executed transactions cannot be reversed, and anyone can deploy hand-crafted compiler-bypassed bytecode. Static verification is thus a single point of failure: one exploited bug can have catastrophic consequences. As a practical security measure, the Move VM implements defense-in-depth *runtime safety checks* that, after static verification, independently verify the critical invariants during execution. These runtime safety checks do not share implementation with the static verifier. Multiple proof-of-concept exploits, surfaced through bug bounty reports and internal audits, bypassed the static verifier but were caught by the runtime checks: the defense in depth working as intended.

In this paper, we describe three groups of runtime checks, each motivated by real-world examples. *Type checks* (§3.1) verify that the dynamically tracked type of every value matches the static type the bytecode expects to consume. *Ability checks* (§3.2) verify that operations such as copy, drop, or store to the global state are only performed on values whose declared abilities permit them. *Reference checks* (§3.3) verify the same freedom from dangling references and aliased mutable access that the static verifier is supposed to enforce, but using fundamentally different representations and algorithms.

We then describe two techniques for reducing the runtime costs of these checks: *trusted code* (§4) that waives them under certain conditions, and an *asynchronous mode* (§5) that leverages idle threads in Block-STM [13] (Aptos's parallel transaction execution engine). We present an evaluation of the performance impact of the checks on real-world benchmarks (§6). Our implementations are openly available [4].

2 Move on Aptos

We briefly summarize key aspects of Move on Aptos [3, 8].

Typing and generics. Move is strongly statically typed. Generic functions and types are instantiated by substitution at the call site. Substitution is carried through transitively: accesses to type-indexed global storage (below) are keyed by concrete types at runtime.

Abilities. Each type carries a subset of the four abilities: `copy`, `drop`, `store`, and `key`. Types without `drop` or `copy` are linear: every value of such a type must be moved, globally stored, or explicitly destructured. `key` indicates a type which can be stored as a global resource, and `store` a type which can be transitively included in resources.

Algebraic data types. Structs are nominal product types; enums are nominal sums whose variants carry payloads. Pattern matching may bind references directly into a variant's fields.

Modular encapsulation. Privileged operations (construction, destruction, and field access) on a private type are confined to its defining module, so module-external code can manipulate values only through module-provided functions.

References. Immutable (`&T`) and mutable (`&mut T`) references obey a static aliasing discipline comparable to Rust's [18, 20]: two live mutable references cannot point to overlapping memory, and references cannot outlive the values they borrow. In contrast to Rust, Move (a) does allow mutable references to global storage, but (b) has no reference lifetimes and does not allow references to be stored in structs or enums or captured by closures.

Global storage. Global state is a map indexed by (resource type, address) pairs. Resources are types with the `key` ability. They are accessed through `T[addr]`, `&T[addr]`, and `&mut T[addr]`.

3 Runtime Safety Checks

This section describes the defense-in-depth runtime checks. Each is motivated by real-world examples of security bugs they are designed to prevent.

3.1 Type Safety

Move has nominal typing: two structs with identical layout but different names are distinct types. A *type-confusion* exploit breaks that distinction, and can be catastrophic when one of the two types is privileged. Consider the example:

```
module 0x1::admin {
 struct Capability(address);
 public fun admin_action(c: &Capability) {
  /* Capability gated action */
 }
}
```

```
module 0xdeadbeef::attacker {
 struct FakedCapability(address);
}
```

The attacker can freely construct their owned type `FakedCapability`; a type confusion exploit would allow passing it to `admin_action` as if it were a `Capability`.

Example. When vector operations were added to the VM during Move's original development at Meta, the bytecode verifier missed an element-type check for push. The compiler performed this check; no input source program could reach the bug. But an attacker could directly construct bytecode, conceptually doing:

```
let v: vector<Capability> = vector::empty();
v.push_back(FakedCapability(victim)); // should be rejected
admin_action(&v[0]);  // runs on a faked type
```

The two types share a layout, so the program runs without corruption, but the gated action runs on a faked type. This bug was found as part of the Aptos bug bounty program before going into production.

Performing the Checks. To prevent type confusion, in addition to the bytecode verifier, type checks are performed at runtime. A type stack is maintained shadowing the operand stack of the VM. The VM's locals are typed, so every value's type is known. When a value is pushed on the operand stack, its type is

pushed on the type stack. When a value is popped, by being stored into a local or used by an operation, its type is popped and compared with the expected type of the local or operation; a mismatch is type confusion and raises a runtime error.

3.2 Ability Safety

Bugs not enforcing `copy` ability are very damaging. A value whose type lacks `copy` is meant to be unique: this is what makes the `Capability` of §3.1 a single-issue permission, and what makes a `Coin` balance impossible to counterfeit. A runtime that produces a fresh value of a no-`copy` type silently duplicates that resource.

Example. Move's ability system permits an assignment to downgrade a value's abilities (i.e., width subtyping over the ability set, which is sound). But due to the classic invariance of mutable references [24], applying this rule through a mutable reference is unsound. When Move was extended with closures, a corner case (see example below) did not enforce invariance. An exploit could capture a no-`copy` resource `r` inside a no-`copy` closure `h`, store that closure into a variable `f` of `copy`-able closure type, and then copy `f` freely, calling it multiple times, duplicating the no-`copy` resource `r`.

```
struct NoCopy(u64) has drop;

public fun assign<T: drop>(
    ref: &mut T, x: T
) {
 // old ref value dropped
 *ref = x;
}
```

```
let r = NoCopy(42);
let h: || has drop = || {
    // capture and use `r`
};
let f: || has drop + copy = || {};
assign<|| has drop>(&mut f, h);
// ^ bug if not rejected: covariant
let g = copy f; f(); g();
```

This bug was found as part of pre-production security auditing.

Performing the Checks. The VM performs runtime checks for violations of each of the abilities `copy`, `drop`, `key`, and `store`. These checks are performed when the corresponding instruction is executed. For `drop`, the VM tracks which values have not been consumed on function exit. The runtime checks for `drop` are more precise than the static checks, as the latter conservatively rejects if a no-`drop` local may not be consumed on some control-flow path (including infeasible paths), while the former only checks the real executed path.

3.3 Reference Safety

The mutable-aliasing rule is critical in the presence of `enum` types: a reference into a variant payload is type-safe only while the enum still holds that variant.

Example. If two mutable references to the same enum can simultaneously be live, one can switch the variant under the other. Consider the following code:

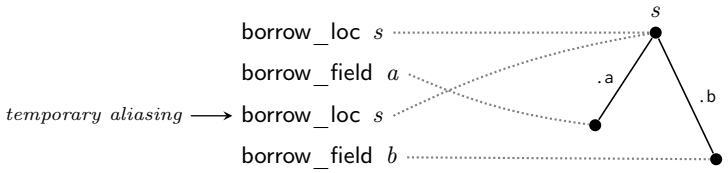

Fig. 1: A dotted line links each bytecode operation (left) to the access-path-tree node (right) for the place its pushed reference denotes.

```
enum E { V1(Capability), V2(FakedCapability) }

fun aliased_swap(a: &mut E, b: &mut E, faked: &mut E) {
    // faked is V2(FakedCapability(victim))
    match (a) {
        E::V1(c) => {
            mem::swap(b, faked); // *a is now V2(..)
            admin_action(c); // c still &Capability
        },
        _ => {},
    }
}
```

A call of the form `aliased_swap(p, q, ...)` where `p` and `q` alias should normally be rejected by the bytecode verifier. However, the implementation of the static borrow analysis had a bug in its *borrow graph* representation. The analysis treats borrows along edges with distinct labels as disjoint, but the representation of labels for enum fields was not canonical: two distinct labels could denote the same field offset, though only in carefully hand-crafted bytecode (not shown here), never in compiler-emitted code. Verification thus admitted such bytecode where `aliased_swap` could be called with aliased mutable references, resulting in the type confusion vulnerability shown in the listing above. This bug was discovered via the bug bounty program after the introduction of enum types and was never exploited on the blockchain.

Performing the Checks. Checking reference safety at runtime for Move-style borrow semantics is novel (§7). A challenge is that when constructing references, mutable reference aliasing *is* allowed. Only once those references are used must aliasing be forbidden. For example, consider the valid code:

```
struct S {a: u64, b: u64}
let s: S;
let r1 = &mut s.a; let r2 = &mut s.b;
*r1 = 1; *r2 = 2; // r1 & r2 reference disjoint fields
```

The corresponding stack-machine bytecode fragment is shown in Fig. 1. While we have a mutable reference to $s.a$ on the stack, we temporarily create a new mutable reference to the whole of s, overlapping with the existing one (Fig. 1). The aliasing is eliminated only when this reference is consumed to create the reference to $s.b$. Thus, a naive approach to runtime reference safety checking, for example via reference counting, would be infeasibly strict.

Our checker instead executes an *abstract shadow semantics* in lockstep with execution. The VM's operand stack, locals, and call frames are mirrored by a shadow state in which every non-reference value is opaque and every reference is

a fresh identifier ρ minted at its borrow site. The shadow state never inspects runtime values or memory addresses; conceptually, it is an abstract interpretation of the single path taken, embedded along with the concrete execution.

State representation. References point into trees of *places*, not at each other. Each function lazily grows *access path trees* rooted at its locals, at the global resource types it borrows, and at the values behind its reference parameters. Edges are field offsets: fields of different enum variants that share an offset share an edge. A reference ρ is a pointer to a node (representing a place) in one of these trees, tagged with its mutability. Every aliasing question reduces to an ancestor/descendant relation between the places two references denote; no borrow genealogy among the references themselves is recorded.

Poisoning, lazy fails. A destructive operation, like a write through a mutable reference or the invalidation of a local by move_loc, poisons every other reference whose places it invalidates. No error is raised at the operation itself. If a poisoned reference is later *used* — read, written, borrowed through, or passed to a callee — a transaction-aborting error is raised. Poisoned references may be moved around or dropped without error. Overlapping references that never destructively interact are never flagged at all, admitting the temporary aliasing illustrated above.

Compositional. The checker is compositional across function boundaries. A call consumes its reference arguments, checks at the call site that the places reachable from each mutable argument are disjoint from those reachable from every other reference argument, and hands the callee fresh trees rooted at its own reference parameters. Passing a mutable reference into a call is a destructive operation, to keep the checker compositional (callee may mutate arbitrarily through it). On return, every returned reference must denote a place inside the tree of some reference parameter; its access path is replayed into the corresponding tree of the caller. Returned mutable references must also denote mutually disjoint places. Hence, the checker tracks precisely which parameter each returned reference derives from, along the executed path, while the static verifier conservatively assumes that a returned reference may borrow from every reference argument.

Thus, the checker realizes a *relaxed dynamic semantics* of the static borrow analysis: (a) bytecode accepted by a correct static verifier is expected to pass the runtime checks on every execution, and (b) the checker admits strictly more behaviors, because it observes only the executed path and has none of the verifier's join and call-summary over-approximations. We have validated (a) empirically — through extensive testing, fuzzing, and re-execution of billions of historical Aptos transactions — without observing a false positive. We have confirmed (b) by hand-crafting bytecode that the static verifier rejects, yet which executes without violating reference safety and is hence admitted by the checker.

For the example (Fig. 1), the bytecode fragment runs successfully: the second borrow_loc s briefly creates a second mutable reference at the tree root for s, but that reference is consumed by borrow_field b before any destructive write fires, so nothing is poisoned.

The exploit described earlier in this section, that creates the aliased mutable references, is caught by poisoning instead: edges in the enum's access path tree

are field offsets, so the two references land on the same node by construction. The destructive `mem::swap` then poisons the still-live payload reference `c` sitting at a descendant place, and the `admin_action(c)` aborts because it is `c`'s use.

4 Trusted Code

Because runtime checks impose a performance overhead, the VM supports *trusted code*: modules for which these checks are waived. Trusted deployment is currently under Aptos Foundation governance and limited to the Aptos framework; each deployment is approved by the Aptos operators, and we are also investigating staking-based models. Unlike arbitrary bytecode published on-chain, trusted code is always compiled from source, and thus passes the compiler's extensive type, ability, and reference safety checks before undergoing static bytecode verification. Defense in depth is therefore preserved: for trusted code, the compiler and the static verifier form the two independent layers, with the compiler playing the role that the runtime checks play for untrusted code.

Executing trusted code disables the checks of §3.1 and §3.2. Because checks like type safety accumulate state during execution, the transitions between untrusted and trusted code must be modeled carefully. For example, trusted code does not maintain the type stack, so when returning from a trusted function the return value's type must be synthesized for the untrusted caller from the type signature.

5 Asynchronous Checking

Aptos executes blocks of transactions in parallel using *Block-STM* [13], an optimistic concurrency-control protocol in which a pool of worker threads speculatively runs transactions out of order and re-validates them against later-published writes; executions that lose a race are aborted and re-executed. Block-STM also tracks when each transaction is committed. If there is not enough parallelism in the block due to read-write conflicts, worker capacity remains unused.

That idle capacity can be used to run the runtime safety checks *asynchronously*. The VM performs no runtime checks but logs a compact *execution trace* for each transaction, containing only control-flow shape. After a transaction commits, a job to replay its trace is scheduled. The replay walks the bytecode along the recorded path and runs the checks of §3.1 and §3.2. Transaction commit latency is unaffected: replay runs only after the commit, proceeding in parallel with the rest of the block's execution. A violation found during replay aborts the block before its writes are persisted on-chain.

Asynchronous checks run on committed transactions only, so speculative executions do not pay their cost. They fail only when a static-verifier bug is being exploited, so an attacker cannot consistently slow the network by triggering such failures.

6 Performance

Our real-world benchmark, derived from Decibel [11], simulates a perpetual-futures DEX with 100 markets, driven by a weighted mix of oracle price updates, bulk market-maker orders, and retail taker orders. It is executed by Block-STM [13] on a 60-vCPU GCP instance with 235 GB of RAM; worker pool varies

Table 1: Throughput (txns/sec, higher is better) under type and ability checking. Slowdown vs. the no-checks baseline in parentheses; "waiver" skips trusted framework code (§4); "asynchronous" defers checks to trace replay (§5).

	baseline	synchronous		asynchronous	
workers		no-waiver	waiver	no-waiver	waiver
2	864	611 (−29.3%)	739 (−14.5%)	513 (−40.6%)	624 (−27.7%)
4	1427	1026 (−28.1%)	1236 (−13.4%)	914 (−35.9%)	1088 (−23.7%)
8	1915	1350 (−29.5%)	1620 (−15.4%)	1427 (−25.5%)	1598 (−16.5%)
16	2019	1427 (−29.3%)	1691 (−16.2%)	1690 (−16.3%)	1799 (−10.9%)
32	1904	1363 (−28.4%)	1621 (−14.9%)	1633 (−14.2%)	1716 (− 9.9%)

Table 2: Throughput (txns/sec) with isolated runtime reference safety checking.

workers	ref checks off	ref checks on	slowdown
2	858	630	−26.5%
4	1430	1054	−26.3%
8	1918	1370	−28.6%
16	2011	1450	−27.9%
32	1893	1375	−27.4%

from 2 to 32 threads. Each reported number is the mean of three runs; run-to-run spread is below 1.3%.

Type and ability checks share their implementation, so Table 1 reports their combined cost (without reference checks). The combined checks cost 28–29% of throughput, uniformly across worker counts. The trusted-code waiver (§4) roughly halves this to 13–16%, as about two-thirds of executed instructions on this workload fall into trusted framework code. The asynchronous mode (§5) trades a small tracing overhead for deferred checking on idle workers: below 8 workers it is slower than synchronous (no idle threads), reaches parity at 8, and at 16–32 brings the cost to 10–11%, the preferred configuration at production concurrency.

The reference checker is a fully separate mechanism with its own shadow state; Table 2 reports it in isolation (type and ability checks disabled): a 26–29% slowdown, uniform across worker counts. Reference checking is fully functional but not yet optimized, and does not yet support the trusted-code waiver or asynchronous deferral; it is therefore not yet live in production. The type-check numbers above set a reasonable expectation for what those mechanisms could recover.

7 Related Work and Conclusion

Static bytecode verification at load time is standard for typed managed VMs: the JVM [19] and CLR [21] check well-formedness, typing, and stack discipline before execution, and sibling Move VMs share Move's verifier. None of these adds a *second*, runtime layer of safety checks: once a module is admitted, its dynamic semantics is trusted. Dynamically typed languages such as JavaScript and Python sit at the opposite end, performing the runtime type checks of §3.1 but lacking the *first* static layer. The EVM [15, 26] and Solana's SVM form a third regime: their untyped bytecode admits neither a static type verifier nor a

runtime type discipline. Aptos's pairing of a static verifier with a complementary runtime layer is thus unique among production VMs.

The runtime reference safety checks (RRSC) we describe are novel. The closest related work, Tree Borrows and Stacked Borrows [17, 25] (TB/SB), clarifies Rust's unsafe aliasing rules, but does not share RRSC's requirement of no false positives over the static verifier. TB/SB also tolerate two or more orders of magnitude of overhead, not being intended for production or large fuzzing workloads; unlike RRSC they maintain provenance genealogies among references and are not function-modular. Also, Move and Rust's reference semantics differ (§2). As future work, the dynamic semantics of the reference checker is also interesting as a model for what reference safety means, since static borrow analysis over-approximates in a method-dependent way and no canonical operational semantics exists; we plan to fully formalize this.

Deferring runtime checks off the execution path and replaying a recorded trace later is a well-known idea: decoupled dynamic analysis [10, 16, 22] and offline runtime verification [6] move expensive monitoring onto spare cores. Our asynchronous checks differ in that the parallel blockchain execution model itself supplies the idle capacity (from workers waiting on read-write conflicts) and runs all checks online, so an offending transaction is detected automatically.

To conclude, we argue the need for runtime safety checks, motivated by real-world bugs, as a second line of defense to the static verifier; they also serve as a semantic, precise oracle for fuzzing the verifier itself. We presented a novel runtime reference safety checker (yet to be launched in production) and two techniques for reducing the cost of runtime checks in a blockchain setting, with the type and ability checks (in production) introducing about 10% overhead on Aptos workloads.

Acknowledgement. Aptos auditor firm OtterSec under lead of Robert Chen highly influenced this work.

Disclosure of Interests. All authors are employees of Aptos Labs and may hold equity in the company. The work described is implemented in the Aptos blockchain.

References

1. Amsden, Z., Arora, R., Bano, S., Baudet, M., Blackshear, S., Bothra, A., Cabrera, G., Catalini, C., Chalkias, K., Cheng, E., Ching, A., Chursin, A., Danezis, G., Giacomo, G.D., Dill, D.L., Ding, H., Doudchenko, N., Gao, V., Gao, Z., Garillot, F., Gorven, M., Hayes, P., Hou, J.M., Hu, Y., Hurley, K., Lewi, K., Li, C., Li, Z., Malkhi, D., Margulis, S., Maurer, B., Mohassel, P., de Naurois, L., Nikolaenko, V., Nowacki, T., Orlov, O., Perelman, D., Pott, A., Proctor, B., Qadeer, S., Rain, Russi, D., Schwab, B., Sezer, S., Sonnino, A., Venter, H., Wei, L., Wernerfelt, N., Williams, B., Wu, Q., Yan, X., Zakian, T., Zhou, R.: The Libra Blockchain. https://developers.libra.org/docs/the-libra-blockchain-paper (2019)
2. Aptos Labs: The Aptos Framework Book. https://aptos-labs.github.io/framework-book/ (2023), accessed: 2026-05-09

3. Aptos Labs: The Move on Aptos Book. https://aptos-labs.github.io/move-book/ (2023), accessed: 2026-05-09
4. Aptos Labs: Aptos Core. https://github.com/aptos-labs/aptos-core (2026), accessed: 2026-06-14
5. Aptos Labs: Aptos Network. https://aptosnetwork.com/ (2026), accessed: 2026-06-13
6. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer (2018)
7. Blackshear, S., Cheng, E., Dill, D.L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Rain, Russi, D., Sezer, S., Zakian, T., Zho, R.: Move: A Language With Programmable Resources (2019), https://developers.libra.org/docs/move-paper
8. Blackshear, S., Dill, D.L., Qadeer, S., Barrett, C.W., Mitchell, J.C., Padon, O., Zohar, Y.: Resources: A safe language abstraction for money (2020), https://arxiv.org/abs/2004.05106
9. Blackshear, S., Mitchell, J.C., Nowacki, T., Qadeer, S.: The Move borrow checker. CoRR **abs/2205.05181** (2022), https://arxiv.org/abs/2205.05181
10. Chow, J., Garfinkel, T., Chen, P.M.: Decoupling dynamic program analysis from execution in virtual environments. In: USENIX Annual Technical Conference (USENIX ATC). pp. 1–14 (2008)
11. Decibel trading engine. https://decibel.trade/ (2026), accessed: 2026-06-14
12. Dill, D.L., Grieskamp, W., Park, J., Qadeer, S., Xu, M., Zhong, J.E.: Fast and reliable formal verification of smart contracts with the move prover. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022). Lecture Notes in Computer Science, vol. 13243, pp. 183–200. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_10
13. Gelashvili, R., Spiegelman, A., Xiang, Z., Danezis, G., Li, Z., Malkhi, D., Xia, Y., Zhou, R.: Block-STM: Scaling blockchain execution by turning ordering curse to a performance blessing. In: Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP 2023). pp. 232–244. ACM (2023). https://doi.org/10.1145/3572848.3577524
14. Grieskamp, W., Zhang, T., Kashyap, V., Silverman, J.: Formal verification of imperative first-class functions in Move. CoRR **abs/2605.10007** (2026), https://arxiv.org/abs/2605.10007
15. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B.M., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A complete formal semantics of the ethereum virtual machine. In: CSF. pp. 204–217. IEEE Computer Society (2018)
16. Jee, K., Kemerlis, V.P., Keromytis, A.D., Portokalidis, G.: ShadowReplica: Efficient parallelization of dynamic data flow tracking. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 235–246 (2013)
17. Jung, R., Dang, H.H., Kang, J., Dreyer, D.: Stacked borrows: An aliasing model for Rust. Proc. ACM Program. Lang. **4**(POPL), 41:1–41:32 (2020)
18. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: Securing the foundations of the Rust programming language. Proc. ACM Program. Lang. **2**(POPL), 66:1–66:34 (2018)
19. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley (1997)
20. Matsakis, N.D., Klock, II, F.S.: The Rust Language, nourl = http://doi.acm.org/10.1145/2692956.2663188. Ada Lett. **34**(3), 103–104 (Oct 2014). https://doi.org/10.1145/2692956.2663188

21. Meijer, E., Wa, R., Gough, J.: Technical overview of the common language runtime (2000)
22. Ming, J., Wu, D., Wang, J., Xiao, G., Liu, P.: StraightTaint: Decoupled offline symbolic taint analysis. In: 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 308–319 (2016)
23. Park, J., Zhang, T., Grieskamp, W., Xu, M., Giacomo, G.D., Chen, K., Lu, Y., Chen, R.: Securing Aptos Framework with Formal Verification. In: 5th International Workshop on Formal Methods for Blockchains (FMBC 2024). Open Access Series in Informatics (OASIcs), vol. 118, pp. 9:1–9:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/OASIcs.FMBC.2024.9
24. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, MA, USA (2002)
25. Villani, N., Dang, H.H., Jourdan, J.H., Jung, R., Dreyer, D.: Tree borrows. Proc. ACM Program. Lang. **9**(PLDI) (2025)
26. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2014), https://ethereum.github.io/yellowpaper/paper.pdf