TasmScan: Continuation-Aware Taint Analysis for TVM Bytecode with Savelist Abstraction

Yixuan Liu
Nanyang Technological University
Singapore, Singapore
LIUY0255@e.ntu.edu.sg

Yin Wu
Xi'an Jiaotong University
Xi'an, China
wuyin@stu.xjtu.edu.cn

Yi Li✉
Nanyang Technological University
Singapore, Singapore
yi_li@ntu.edu.sg

Abstract

The Open Network (TON), with a peak market capitalization exceeding $20 billion and over 175 million activated on-chain addresses, relies on the TVM (TON Virtual Machine) to execute smart contracts. TVM uses first-class continuations with *savelist*s to manage control flow and register state across continuation invocations. Since *savelist*-captured registers allow data to flow across continuation boundaries without passing through the operand stack, bytecode-level analyses cannot construct complete data flow tracking without explicitly modeling *savelist* semantics. We present TasmScan, the first bytecode-level static analysis framework for TVM that enables cross-continuation data flow reasoning without requiring source code. TasmScan models *savelist* semantics via forward register analysis with a formal over-approximation guarantee for exact-resolved save sites and locally tracked register definitions, then lifts bytecode into TASIR, a typed intermediate representation, and performs path-sensitive taint analysis with context-aware sources to detect defects. We evaluate TasmScan on 2,921 contracts from the TON verifier registry and a labeled benchmark of 208 contracts with human-confirmed ground truth. On the full corpus, TasmScan resolves 294,546 dynamic continuation targets with 100% precision; ablation confirms that *savelist* propagation is essential for resolving indirect register calls that depend on cross-continuation register passing. On the benchmark, TasmScan detects 95.3% of defects across five classes with 96.8% precision. A 366-pair stratified sample from the full corpus estimates 85.8% overall precision. TasmScan offers a 17× median speedup over the state-of-the-art symbolic-execution baseline, and in the path-analysis comparison completes 100% of analyses with zero crashes or timeouts.

CCS Concepts

• **Software and its engineering** → **Formal software verification**; • **Security and privacy** → **Software security engineering**.

Keywords

smart contract security, static analysis, abstract interpretation, TVM, continuations

ACM Reference Format:
Yixuan Liu, Yin Wu, and Yi Li. 2026. TasmScan: Continuation-Aware Taint Analysis for TVM Bytecode with Savelist Abstraction. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834346

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834346

1 Introduction

Smart contracts deployed on blockchain platforms manage substantial financial assets, making their security a critical concern [4, 24]. The Open Network (TON), initially designed by Nikolai Durov [12] to support Telegram's large user base, has grown into a major decentralized platform with a peak market capitalization exceeding $20 billion and over 175 million activated on-chain addresses [31]. Its DeFi ecosystem has reached a total value locked (TVL) of up to $800 million [11], hosting thousands of smart contracts for token transfers, lending protocols, NFT marketplaces, and governance mechanisms [29]. A recent empirical study found that approximately 94% of TON contracts contain at least one defect [26], underscoring the need for automated analysis to safeguard the significant assets under management.

TON smart contracts are compiled to bytecode and executed on the TVM. Unlike the Ethereum Virtual Machine (EVM), which uses address-based jumps, TVM employs *first-class continuations* with *savelist*-mediated register save/restore as its sole control-flow primitive [23]. Figure 1 illustrates this mechanism: `CALLX` (blue) implicitly saves register `c0` into the callee's *savelist*, and a later `JMPX` (amber) jumps to the restored value. Without *savelist* modeling, the jump target is unresolved, leaving a gap in the control-flow graph (CFG). The red dashed path further shows how taint propagates across continuation boundaries through the same *savelist* mechanism.

Analyzing TON smart contracts poses three challenges. *First*, source code is often unavailable for deployed contracts, so the analysis must operate directly on bytecode; TONScanner [26] provides source-level analysis but cannot reach bytecode-level *savelist* behaviors. *Second*, data flows across continuation boundaries through register save/restore operations, a pattern invisible to analyses designed for address-based control flow [5, 9, 14, 32]. *Third*, statically abstracting *savelist* effects across all execution paths is difficult; TSA [13] faithfully executes *savelist* operations per path via symbolic execution but models each *savelist* as a concrete structure with no join or widening, leaving cross-continuation data flow unabstracted.

To address these challenges, we propose TasmScan, which directly analyzes deployed TVM bytecode. TasmScan first resolves continuation targets via stack simulation to recover the CFG, then applies forward register analysis to statically reconstruct each continuation's *savelist* effects, with a soundness guarantee scoped to exact-resolved save sites and locally tracked register definitions. Built on this foundation, TasmScan lifts bytecode into a typed IR

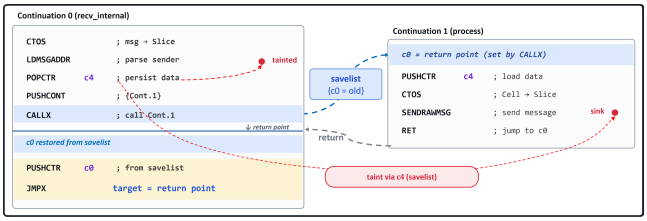

Figure 1: Savelist-dependent indirect jump and cross-continuation taint flow.

(TASIR) and performs continuation-aware taint analysis to detect 5 defect classes. Evaluation on 2,921 real-world contracts shows that TasmScan is both effective and efficient compared with the state-of-the-art symbolic-execution baseline.

In summary, this paper makes the following contributions:

- **Formal savelist abstraction.** We formulate a set-valued dataflow model for TVM *savelist* semantics with a tracked-definition soundness theorem under exact-resolved save sites, enabling cross-continuation data flow reasoning at the bytecode level while making the proof scope explicit. (Sect. 3.4)
- **TASIR and continuation-aware taint analysis.** We design TASIR, a typed intermediate representation for TVM bytecode, and implement continuation-aware taint analysis on top of it to detect 5 defect classes in TVM/TON smart contracts. (Sects. 3.3, 3.5 and 3.6)
- **Comprehensive evaluation.** We evaluate TasmScan on 2,921 contracts from the TON verifier registry. TasmScan resolves 294,546 dynamic continuation targets with 100% precision, including 1,028 *savelist*-dependent targets across 129 contracts that require cross-continuation register tracking. It achieves a 95.3% detection rate with a 17× median speedup over TSA and 100% analysis completion with zero crashes. (Sect. 4)

2 Background

2.1 The TON Blockchain

The TON blockchain [12, 29] is a multi-chain platform using asynchronous message passing between smart contracts. Developers write contracts in *FunC* (a C-like language), *Tact* (a higher-level language), *Tolk* (a TypeScript-like language), or directly in *Fift* (a low-level stack language); all compile to TVM bytecode, serialized in the *Bag of Cells* (BOC) format and deployed on-chain. Since source code is not always available for deployed contracts, bytecode-level analysis is essential.

TON contracts define standard entry points dispatched by a selector value on the stack. The primary entry point, `recv_internal` (selector 0), handles messages from other contracts. The initial stack provides the selector (`s0`), message body (`s1`), and message cell (`s2`) as inputs to the contract logic.

Table 1: Architectural comparison between EVM and TVM.

Aspect	EVM	TVM
Control flow	JUMP/JUMPDEST	First-class continuations
State transfer	None	Savelist (register snapshot)
Data model	256-bit words	Cell/Slice/Builder tree
Message format	ABI encoding	TL-B encoding
Function dispatch	4-byte selector	Dictionary dispatch
Gas model	Prepaid	Credit-based gas

2.2 TVM and Continuations

The EVM uses program-counter-driven control transfer (`JUMP`/`JUMPI`) [36]. The TVM [23] replaces this model with *first-class continuations*: each continuation is a composite value containing executable code and a *savelist*, a partial mapping from register indices to values. This mechanism serves as the sole control-flow primitive in TVM.

The TVM is a stack-based virtual machine with 16 control register slots (`c0`–`c15`), of which seven have defined roles, and a tree-structured data model based on *Cells* (up to 1,023 bits of data and 4 references to other Cells), *Slices* (read cursors), and *Builders* (write cursors). Four control registers are central to our analysis: `c0` holds the return continuation, `c1` the alternative-return continuation, `c3` the method dictionary, and `c4` stores persistent contract data. The remaining slots serve roles such as exception handling (`c2`), output action accumulation (`c5`), and blockchain context (`c7`) [23]. We write $Reg = \{c_0, c_1, c_2, c_3, c_4, c_5, c_7\}$ for the set of active control registers.

When control transfers between continuations, register values can be preserved through a mechanism called the *savelist*:

Definition 1 (TVM Continuation). *An ordinary TVM continuation is a tuple $c = \langle code, \ell \rangle$ where code is an instruction sequence and $\ell : Reg \rightharpoonup Value$ is a partial function mapping register indices to values (the savelist).*[1]

The `SAVE` r instruction writes the current value of register r into the *savelist* of the return continuation (`c0`). When a continuation is invoked (*e.g.*, via `EXECUTE`, `CALLX`, `JMPX`, or conditional instructions), its *savelist* entries are restored: for each register index r in the domain of ℓ (i.e., each register that was explicitly saved), r is set to $\ell(r)$. This mechanism enables data to flow across continuation boundaries through register save/restore operations, a pattern that requires dedicated static analysis support.

Table 1 summarizes the key architectural differences between EVM and TVM that motivate the design of TasmScan.

3 Approach

Figure 2 shows the TasmScan pipeline. BOC disassembly (Sect. 3.1) decodes BOC-serialized bytecode into typed instructions with stack-effect annotations. Continuation resolution (Sect. 3.2) performs

[1] Runtime continuations carry additional fields (stack, codepage, argument count) [23]; the code and *savelist* are the fields relevant to cross-continuation data flow.

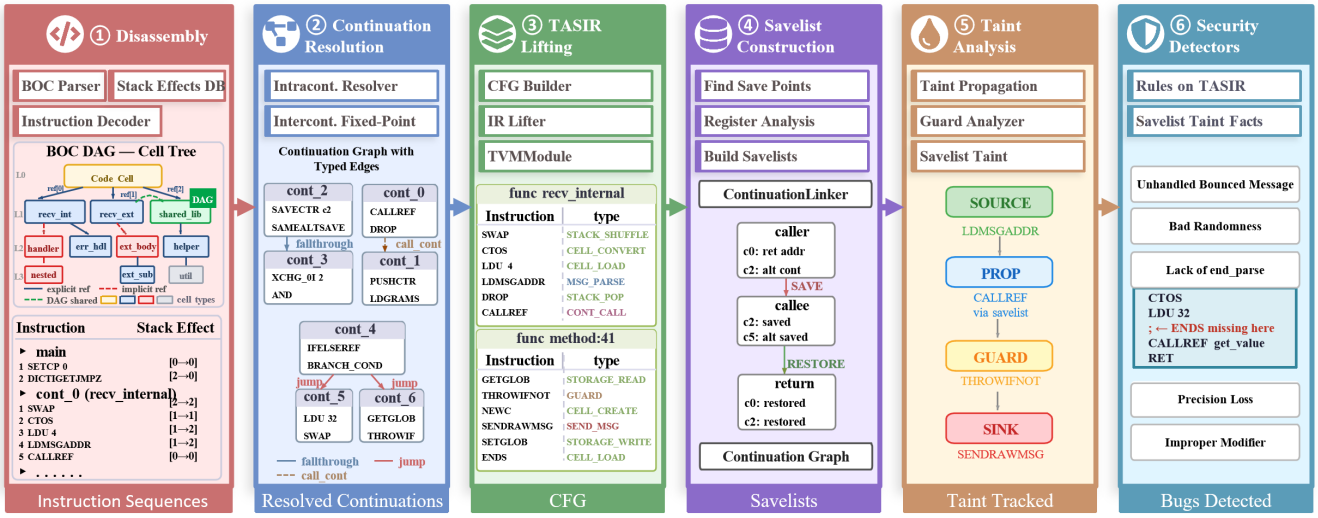

Figure 2: Overview of TasmScan.

stack simulation to resolve dynamic continuation targets and build the CFG. TASIR lifting (Sect. 3.3) translates resolved instructions into a typed intermediate representation. Savelist construction (Sect. 3.4) statically models register save/restore effects across continuation boundaries. Taint analysis (Sect. 3.5) is path sensitive and propagates taint across continuation boundaries via savelists. Security detectors (Sect. 3.6) identify defects from taint findings and structural patterns.

3.1 BOC Disassembly

The disassembly stage converts BOC-serialized TVM bytecode into typed instructions annotated with stack effects.

Instruction database. Our disassembler's opcode database covers the TVM instruction set with encoding, mnemonic, and interval stack-effect annotations, derived from the official specification [23, 29].

Variable-length decoding. TVM opcodes are variable-length (8–24 bits). The decoder performs prefix matching over sorted opcode ranges: it aligns the remaining slice to a 24-bit window, identifies the next opcode, and extracts its operands.

All-or-nothing fallback. When decoding fails mid-slice (*e.g.*, in data cells), the decoder discards the partial result and emits the entire slice as raw data, avoiding spurious instructions. The raw payload remains attached to the BOC cell tree but is not treated as executable input by CFG construction or later analyses.

Stack effect annotations. Each decoded instruction carries an interval effect $[in_{\min}, in_{\max}] \rightarrow [out_{\min}, out_{\max}]$, used by continuation resolution and the detectors. Algorithm 1 splits the BOC root into the main code body and an optional method dictionary in `c3`, then decodes each entry into a linear instruction sequence.

Algorithm 1: TVM Bytecode Disassembly

Input: BOC bytecode B
Output: Method-indexed instruction sequences $\mathcal{I}$

```
(code, dict) ← SplitCodeAndDict(ParseBOC(B).root);
I ← {0 ↦ DecodeSlice(code)};
if dict ≠ ⊥ then
    forall (key, val) ∈ ParseDict(dict) do
        I[key] ← DecodeSlice(val);
return I;
```

3.2 Continuation Resolution

Given the instruction sequence produced by the disassembler, this stage resolves which continuation each branch, call, or jump instruction targets. Since TVM control-transfer instructions consume continuations from the stack, determining the target requires static reasoning about stack contents. We perform lightweight stack simulation using a three-category lattice (Def. 2).

Example. Consider the two control-transfer sites in Fig. 1. At the blue `CALLX` site, the preceding `PUSHCONT` pushes Cont.1. The stack therefore holds the known continuation token cont(*Cont.1*), and the resolver yields an *exact* target. By contrast, at the `JMPX` site (amber), the operand comes from `PUSHCTR c0`, whose value was restored via the *savelist* after the callee returned. The register map yields unknown for c_0 under a purely local simulation, leaving the target *unresolved* until *savelist* propagation (Sect. 3.4) recovers the binding. This gap motivates the intercontinuation fixed-point described below.

3.2.1 State Representation.

Definition 2 (Stack Token). *A token t is one of: (1)* cont(id) *— a known continuation with identifier id; (2)* val(n?) *— a value (optionally a known constant n); or (3)* unknown *— an unresolvable entry.*

Definition 3 (Analysis Context). *Let Tok be the set of stack tokens (Def. 2). An analysis stack is a pair $\sigma = \langle p, u \rangle$ where $p \in Tok^*$ is a finite sequence of tokens (the tracked stack prefix) and $u \in \mathbb{B}$ (Booleans) is the* `unknown_below` *flag, indicating whether unmodeled elements may exist below that prefix; when u is set, accesses beyond p conservatively yield* unknown. *A register map is $\rho : \{c_0, c_1, c_2, c_3\} \to Tok_\perp$, where $Tok_\perp$ extends tokens with $\perp$ (undefined); only the control-flow registers* `c0`–`c3` *are tracked, as they determine continuation targets. A register read yields $\rho[r]$ when defined and* unknown *otherwise. An analysis context is $\chi = \langle \sigma, \rho \rangle$.*

3.2.2 Intracontinuation Analysis. Algorithm 2 outlines the intracontinuation resolution procedure. Given an instruction sequence and an initial analysis context χ_0, it scans each instruction in order, performing three actions per step: resolving continuation targets from the current state (Line 5), updating the analysis context (Line 9), and emitting transfer edges for intercontinuation propagation (Line 10). The procedure produces four outputs: resolved targets $\mathcal{M}$, exact save-target facts $\mathcal{S}$, unresolved sites $\mathcal{U}$, and callee transfer facts $\mathcal{X}$.

Target resolution. At each control-transfer instruction, the resolver reads the analysis context to determine the jump target (Lines 5–6); control-transfer instructions that yield no target are recorded as unresolved in $\mathcal{U}$. The resolution uses an instruction-specific specification: calls and jumps read the stack top; unary conditionals read the continuation below the condition; multi-target branches (*e.g.*, `IFELSE`) read the required k continuation positions. Each resolved fact carries a confidence tag $\kappa \in \{exact, heur\}$: facts derived directly from the analysis context or from exact method-dictionary lookups are *exact*; recoveries from the dynamic solver are *heur*. When the current instruction is a save operation and the register read of c_0 yields a known continuation, an exact save-target fact $\mathcal{S}[i] = id$ is recorded (Line 8).

State transfer and propagation. The step function (Line 9) applies exact rules for pushes, register reads/writes, and stack permutations. For instructions whose stack behavior is not modeled exactly, the interval stack-effect annotation $[in_{\min}, in_{\max}] \to [out_{\min}, out_{\max}]$ from Sect. 3.1 pops the known minimum, sets the `unknown_below` flag when additional hidden operands may be consumed, and pushes unknown outputs. Finally, typed transfer edges and callee contexts are emitted for intercontinuation propagation (Line 10).

Known-Prefix Fallback. When a control-transfer instruction cannot resolve its continuation via positional access (because the target depth exceeds the known prefix), a fallback scans the entire known prefix for cont tokens. If exactly k continuations are found and the instruction requires exactly k targets, they are returned as the resolution. An exact-count guard prevents false matches when extra continuations are present.

3.2.3 Intercontinuation Fixed-Point. The intracontinuation resolution of Alg. 2 analyzes one continuation body in isolation, so callees start with no knowledge of the caller's stack or registers. To recover caller-derived state, we introduce an intercontinuation fixed-point worklist (Alg. 3) that repeatedly invokes Alg. 2 while propagating entry contexts across continuation boundaries until convergence. Each public entry point is seeded with an initial analysis context (Line 4): the standard TON stack shape for the main entry, or the conservatively widened shape observed at method-dictionary dispatch sites. The main loop dequeues each continuation and invokes the intracontinuation resolution of Alg. 2 (Line 8). The propagated callee context is merged into the target's entry state via the join operator $\sqcup$ (Lines 11–14). The join widens stack prefixes element-wise (mismatched positions become unknown) and applies *unanimity merging* to registers: a register value is retained only when all callers agree, and discarded otherwise. This correctly propagates globally unique bindings such as the method dictionary c_3 while preventing false propagation of caller-specific values such as return addresses in c_0. If the merged entry state changes, the target is re-enqueued, driving iteration to a fixed point. Termination follows because the

Algorithm 2: Intracontinuation Resolution

```
1  function ResolveLocal(I = ⟨I_0, ..., I_{n-1}⟩, χ_0);
   Output: Control-transfer facts M; exact save-target map S;
           unresolved set U; transfer facts X
2  M, S, U, X ← ∅;
3  χ ← χ_0;
4  for i = 0 to n − 1 do
5      T ← ResolveTargets(χ, I_i);
6      if NeedsTarget(I_i) ∧ T = ∅ then U ← U ∪ {i};
7      M[i] += T;
8      if SaveOp(I_i) ∧ ReadReg(χ.ρ, c_0) = cont(id) then
         S[i] ← id;
9      χ' ← Step(χ, I_i);
10     X += TransferEdges(χ, χ', I_i, T);
11     χ ← χ';
12 return M, S, U, X
```

Algorithm 3: Intercontinuation Fixed-Point Resolution

```
   Input: Continuation contexts C; entry points E
   Output: Control-transfer facts M; exact save-target map S;
           unresolved set U
1  ctx[c] ← ⊥ for all c ∈ C;            // Analysis contexts
2  out[c] ← (∅, ∅, ∅, ∅) for all c ∈ C;
3  forall e ∈ E do
4      ctx[e] ← InitCtx(e);
5  W ← E;
6  while W ≠ ∅ do
7      c ← W.dequeue();
8      (M_c, S_c, U_c, X_c) ← ResolveLocal(Body(c), ctx[c]);
9      out[c] ← (M_c, S_c, U_c, X_c);
10     forall (c', χ_t) ∈ X_c do
11         ctx' ← ctx[c'] ⊔ χ_t;
12         if ctx' ≠ ctx[c'] then
13             ctx[c'] ← ctx';
14             W.enqueue(c');
15 (M, S, U) ← Aggregate(out);
16 (M, S, U) ← PruneUnreachable(M, S, U, E);
17 return M, S, U
```

tracked stack prefix, token categories, register domain, and continuation set are finite; $\sqcup$ is monotone and can only replace information with unknown or discard a non-unanimous register value. Thus each entry context changes finitely many times. After convergence, the final intracontinuation facts are aggregated (Line 15), and a depth-first reachability traversal from entry points prunes unresolved edges originating from unreachable code (Line 16), as these represent dead continuations extracted from data cell references (*e.g.*, PUSHREFSLICE) rather than executable code. SAVE-family instructions whose pre-save c_0 binding is not exact are omitted from $\mathcal{S}$ and hence from the savelist construction of Sect. 3.4.

Dynamic Target Solver. When continuation resolution leaves a target unresolved, a fallback solver tries four common patterns before emitting dynamic: direct dictionary dispatch from a preceding PUSHINT, exact PUSHCTR c3; EXECUTE recovery, a weaker method-ID hint from PUSHINT, and BLESS; EXECUTE. Only the exact c3-based recovery contributes to save-target attribution and Thm. 1; heuristic recoveries are used only to improve later CFG coverage. Each call/jump site is then annotated with the target's *savelist* summary rather than recursively inlining callees.

3.3 TASIR: Intermediate Representation

Raw TVM instructions expose numerous encoding variants with identical semantics. Writing analyses directly over these encodings would duplicate transfer functions and detector rules. Our lifter therefore normalizes them into semantic node kinds in TASIR, a typed intermediate representation, while retaining the original TVM instruction in each node for traceability. Using the outputs of Alg. 3, $\mathcal{M}$ drives the construction of a typed CFG that can be reused by downstream analyses, while $\mathcal{S}$ seeds the savelist modeling described in Sect. 3.4.

Each TASIR instruction carries the original opcode and a *kind* that classifies its behavior, such as CONT_CALL, STACK_PUSH, or CELL_LOAD. It also carries the interval stack effect from Sect. 3.1 and optional security labels. Examples include MESSAGE_SEND and AUTH_CHECK. The kind field drives transfer-function dispatch in the taint analysis (Sect. 3.5), and labels identify sinks and sources. Because each node retains its original instruction and exposes an ordinary typed CFG, TASIR can also support analyses beyond the detectors in this paper.

Definition 4 (TASIR Module)**.** *A TASIR module is a tuple* $\Pi = \langle F, D, Edges \rangle$ *where* F *is the set of per-continuation functions,* D *maps continuation identifiers to descriptors, and Edges is the set of typed cross-function edges (call, jump, return, exception). Each continuation descriptor* $d \in D$ *records the continuation's identifier, entry block, basic blocks, and a savelist summary* $\hat{\ell}$ *populated by the savelist construction of Sect. 3.4.*

The *CFG Builder* consumes $\mathcal{M}$ and constructs a CFG with typed edges (fallthrough, branch, call, call_return); call-return edges are emitted only when the c_0 binding is known exactly, as the transfer facts already carry these bindings. TASIR organizes the resulting basic blocks into per-continuation functions and attaches each function to its descriptor in D. The *Stack Analyzer* then computes per-block stack height bounds, and the *Event Extractor* classifies security-relevant effects (*e.g.*, message sends, gas commitment, persistent storage writes) for the detectors in Sect. 3.6. This hierarchy makes the relevant objects explicit: the savelist construction (Sect. 3.4) iterates over functions $f \in F$ for forward analysis and over continuation descriptors $d \in D$ when attaching savelist summaries and propagating them to call sites.

3.4 Static Savelist Modeling

Continuation resolution (Sect. 3.2) determines *where* control flows, while savelist construction determines *what data* crosses those continuation boundaries. We keep them separate because savelist analysis depends on resolved edges but uses a stronger register-domain abstraction.

3.4.1 *Formal Definitions.*

Definition 5 (Tracked Value)**.** *A tracked value is a tuple*

$$\hat{v} = \langle src : Source,\ def : \mathbb{N} \rangle$$

where src is the provenance (e.g., STORAGE, MSG_SENDER) and def is the definition site (instruction index) of a locally tracked *register write within the currently analyzed continuation body.*

Definition 6 (Register State)**.** *Let* $\hat{V}$ *denote the set of all tracked values (Def. 5). A register state is a total function* $\hat{\rho} : Reg \to \mathcal{P}(\hat{V})$*, where* $\mathcal{P}$ *denotes the power set. For a register* r*, the set* $\hat{\rho}(r)$ *contains the locally tracked definition sites that may reach* r *at the current program point.* $\hat{\rho}(r) = \emptyset$ *means that the current value of* r *is either not locally defined on the explored path or has been invalidated by an analysis barrier (e.g., a dynamic-register write).*

Definition 7 (Savelist Summary)**.** *A savelist summary is a pair* $\hat{\ell} = \langle saved, defs \rangle$ *where* $saved \subseteq Reg$ *records which registers may be explicitly saved for the continuation, and* $defs : Reg \to \mathcal{P}(\hat{V})$ *maps each register to the set of locally tracked definition sites that may flow into that saved entry. For a save of register* r *with reaching-definition set* $\hat{R}_r$*, the update is:* $saved' = saved \cup \{r\}$ *and* $defs'(r) = defs(r) \cup \hat{R}_r$*.*

3.4.2 *Construction Algorithm.*

Algorithm 4 formalizes the construction in three phases, operating over the tracked values (Def. 5), register states (Def. 6), and savelist summaries (Def. 7). The first phase (Lines 1–6) collects all save-instruction sites into SP, where $I.idx$ denotes the position index of instruction I: for each save instruction with statically known saved registers, the saved-register set is extracted and the exact save-target map $\mathcal{S}$ supplies the continuation bound to c_0 at that site. The second phase (Line 7) computes Φ, the least fixed point of reaching definitions over the register-state lattice of Def. 6: modeled register writes attach provenance, non-write instructions propagate states conservatively, and CFG joins use pointwise set union. Lines 7–15 build each continuation's *savelist* summary by recording both the registers that may be saved and the tracked definitions that may populate those saved entries.

Example (Fig. 1). Table 2 traces Alg. 4 on the motivating example: CALLX in Cont.0 triggers a save of c0 into Cont.1's *savelist* (Phase 1), reaching-definition analysis identifies the saved value as the return-point continuation (Phase 2), and the *savelist* summary is built accordingly (Phase 3). With this summary, the previously unresolved PUSHCTR c0; JMPX (amber in Fig. 1) can be resolved: the target is the return-point continuation, recovered via the constructed savelist. The red dashed path in Fig. 1 further shows how

Algorithm 4: Savelist Construction

Input: TASIR module $\Pi = \langle F, D, Edges \rangle$; exact save-target map $\mathcal{S}$
Output: Enhanced module Π' with populated savelists

```
1  SP ← ∅;                                              // Save points
2  forall instruction I in functions of F do
3      if SaveOp(I) ∧ I.idx ∈ dom(S) then
4          R_s ← SavedRegs(I);
5          t ← S[I.idx];
6          SP[t] += {(I.idx, R_s)};
7  Φ ← ReachDefs(Π);      // Reaching-definition fixed point
   // Savelist build
8  forall continuation descriptor d ∈ D do
9      d.ℓ̂.saved ← ∅;
10     forall r ∈ Reg do
11         d.ℓ̂.defs[r] ← ∅;
12     forall (i_s, R_s) ∈ SP[d.id] do
13         forall r ∈ R_s do
14             d.ℓ̂.saved ← d.ℓ̂.saved ∪ {r};
15             d.ℓ̂.defs[r] ← d.ℓ̂.defs[r] ∪ Φ[i_s][r];
16 return Π'
```

Table 2: Savelist construction walkthrough for Fig. 1.

Ph.	Action	Input	Output
1	Collect save	CALLX at i_s	$SP[\text{C1}] = \{(i_s, \{c_0\})\}$
2	ReachDefs	$\Phi[i_s][c_0]$	{ret-point}
3	Build $\hat{\ell}$	C1 descriptor	$saved = \{c_0\}$, $defs[c_0] =$ {ret-pt}

taint from LDMSGADDR propagates through c4 via this savelist mechanism to reach SENDRAWMSG in Cont.1.

3.4.3 Soundness Scope. For a concrete value v, let $\text{def}(v)$ denote the instruction index that produced v at runtime, and $\text{dom}(f)$ the domain of a partial function f. For a tracked value $\hat{v}$, let $\gamma(\hat{v}) = \{v \in Value \mid \text{def}(v) = \hat{v}.def\}$; the *src* tag refines precision but does not participate in the soundness argument. We extend γ pointwise to sets of tracked values.

Theorem 1 (Tracked-Definition Soundness of Savelist Construction)**.** *Let $\hat{\ell}_c = \langle saved_c, defs_c \rangle$ be the savelist summary computed by Alg. 4 for continuation descriptor c (notation follows Def. 7). Consider any concrete execution trace π and any concrete continuation instance κ of code c whose savelist is updated in π by a* SAVE*-family instruction whose target continuation is* exact-resolved *in the save-target map $\mathcal{S}$ returned by Alg. 3 and whose saved-register set is statically known. Let ℓ^{π}_{κ} be the concrete savelist carried by that instance after those updates. Then:*

(1) $\text{dom}(\ell^{\pi}_{\kappa}) \subseteq saved_c$ *(saved-register coverage), and*

(2) $\forall r \in \text{dom}(\ell^{\pi}_{\kappa})$*: if the concrete value $\ell^{\pi}_{\kappa}(r)$ is* locally tracked *at the corresponding save site, then $\ell^{\pi}_{\kappa}(r) \in \gamma\big(defs_c(r)\big)$ (tracked-value over-approximation).*

A value is locally tracked when its last reaching write before the save site is a modeled RegWrite *in the analyzed function and no subsequent analysis barrier invalidates that register. Save sites with heuristic or unresolved continuation targets, or with dynamic saved-register operands, are conservatively excluded from the precise value-attribution part of the theorem.*

Composition with continuation resolution. The theorem relies on Alg. 2 for the exact save-target fact $\mathcal{S}[i_s] = c$ and the statically known saved-register set at i_s. Heuristic targets, dynamic targets, and dynamic saved-register operands are excluded from the precise value-attribution claim and remain only in the implementation's conservative fallback.

Proof sketch. Let L be the register-state lattice of Def. 6, ordered pointwise by subset. Because the module and $\hat{V}$ are finite, L is finite. The ReachDefs transfer is monotone: modeled writes install singleton definition sets, the non-write transfer preserves or clears information conservatively, and joins use pointwise union. Thus ReachDefs computes the least fixed point for locally tracked register writes.

For any reachable save site i_s and register r, $\Phi[i_s][r]$ contains every locally tracked definition that may reach r immediately before the save. For Property 1: by the theorem's precondition the saved-register set is statically known, so the concrete registers saved at runtime equal the register set encoded by the save instruction at i_s. Lines 1–6 record exactly these registers into SP, and Lines 7–15 union them into $saved_c$; hence $\text{dom}(\ell^{\pi}_{\kappa}) \subseteq saved_c$. For Property 2: when the concrete value of r is locally tracked (i.e., its last write is a modeled RegWrite not invalidated by an analysis barrier), $\Phi[i_s][r]$ includes that definition by the least fixed-point guarantee. Lines 7–15 union the reaching-definition sets from all save sites targeting c into $defs_c(r)$, so the concrete value is in $\gamma(defs_c(r))$.

3.5 Path-Sensitive Taint Analysis

With the TASIR module and populated savelists, this stage tracks how attacker-controlled data propagates through the program, including across continuation boundaries via savelist-restored registers.

3.5.1 Taint State.

Definition 8 (Taint Lattice)**.** *The taint lattice is:*

$$\hat{d} = \langle src : Source,\ tainted : \mathbb{B},\ checked : \mathbb{B},\ origins : \mathcal{P}(\mathbb{N}) \rangle$$

where origins tracks the definition sites from which taint propagated, and Source is one of: MSG_SENDER, MSG_BODY, MSG_VALUE, CONSTANT, COMPUTATION, STORAGE, or UNKNOWN.

Source tags form a flat order (all tags are incomparable; their join is UNKNOWN). Taint values are ordered componentwise: source tags use that flat order, *tainted* uses the usual Boolean order, *checked* uses the reverse Boolean order (so unchecked is higher), and *origins* uses subset. Merge points keep the source tag when both sides agree and use UNKNOWN otherwise; they OR the taint bit, AND the checked bit, and union the origin sets.

Table 3: Taint transfer function categories.

Category	Kind	Rule
Uncond. source	Msg-field read	$\hat{d}_{out}.tainted \leftarrow true$
Cond. source	Generic load	Taint only if input is msg-derived
Propagation	Data transform	$\hat{d}_{out} \leftarrow \hat{d}_{in}$
Sink	Sensitive op	Report if tainted ∧ unchecked

The analysis state $\hat{S}$ stores four components: a bounded stack of up to K_s taint values, an *overflow* bit, a register map from *Reg* to $\hat{d}_\perp$, and a partial guard map from definition sites to guard instructions. Only the top K_s stack entries are tracked explicitly; deeper pushes discard the deepest tracked entry and set *overflow* = *true*. K_s is a fixed implementation parameter.

3.5.2 Transfer Functions. We define four categories of taint transfer rules (Table 3).

Unconditional sources always produce tainted output because they read attacker-controlled message fields (sender address, value, body). *Conditional sources* taint their outputs only when the input originates from an incoming message, not from persistent storage (c4); this distinction prevents over-tainting of contract-internal data. *Propagation* instructions inherit the input taint unchanged; for multi-output instructions, each output applies its own inheritance rule. *Sinks* do not transform taint but trigger a finding when an operand is tainted and unchecked.

Guard analysis. When a guard instruction (`IF`, `THROWIF`, etc.) is encountered and its condition depends on a tainted value, the corresponding *checked* bit is set, suppressing further reports for that value at downstream sinks.

3.5.3 Cross-Continuation Taint via Savelist. Without savelist-aware propagation, taint information is lost at continuation boundaries (Fig. 1), causing the analysis to miss vulnerabilities that involve data flowing through saved registers. To bridge this gap, at each continuation call/jump instruction I, TasmScan retrieves the precomputed savelist summary (Sect. 3.4) $\hat{\ell} = \langle saved, defs \rangle$. For each saved register $r \in saved$ and each locally tracked definition $\hat{v} \in defs(r)$, if $\hat{v}$ is tainted in the current state, that taint is propagated from definition site $\hat{v}.def$ to the call/jump site $I.idx$:

$$\forall r \in saved,\ \forall \hat{v} \in defs(r) :\ tainted(\hat{v}.def) \implies propagate(\hat{v}.def, I.idx) \tag{1}$$

Here $tainted(i)$ is shorthand for "the current taint state contains a tainted value whose origin/definition site is i," and $propagate(i, j)$ denotes the cross-continuation taint fact forwarded from definition site i to call/jump site j.

Upon entering the target continuation, the *savelist* entries are restored into the register state. The restore-side transfer function updates the taint analysis state accordingly:

$$\forall r \in saved :\ \hat{S}'.regs[r] \leftarrow \mathsf{RestoreSet}(\hat{S}, r, defs(r)) \tag{2}$$

where the restore helper joins the taint states for all tracked definitions in $defs(r)$; when no tracked definitions are available for r, it materializes a placeholder for a saved-but-unattributed register. Thus Eqs. (1) and (2) form the cross-continuation bridge XContTaint used by Alg. 5.

Algorithm 5: Path-Sensitive Taint Analysis

Input: TASIR module Π' with savelists; initial entry pairs $\mathcal{E}_0 = \{(B, \hat{S})\}$ where B is a basic block and $\hat{S}$ its taint state

Output: Taint results $\mathcal{T}$

```
1  T, V ← ∅;                          // Findings; visited (block, fingerprint) pairs
2  W ← PriorityQueue(E_0);            // Seed with entry blocks
3  while W ≠ ∅ do
4      (B, Ŝ) ← W.dequeue();
5      fp ← Fingerprint_{K_s}(Ŝ);
6      if (B.id, fp) ∈ V then continue;
7      V ← V ∪ {(B.id, fp)};
8      forall instruction I in B do
9          Ŝ ← TaintStep(I, Ŝ);          // Transfer function
10         if I.kind ∈ {CONT_CALL, CONT_JUMP} then
11             Ŝ ← XContTaint(Ŝ, I, I.ℓ̂);   // Target's savelist
12         if Sink(I) ∧ Unchecked(Ŝ, I) then
13             T ← T ∪ {(I, Ŝ)};
14     forall successor B' of B do
15         if BackEdge(B, B') ∧ loops(B') > MaxUnroll then
16             T ← T ∪ LoopSummary(B', Ŝ);
17         else
18             W.enqueue(B', Ŝ, priority(B'));
19 return T
```

3.5.4 Worklist Algorithm. The path-sensitive analysis is driven by the priority worklist of Alg. 5. Public entries are seeded with the standard TON entry stack from Sect. 2.1: selector as an untainted constant, message body/message cell/message value as message-derived sources, contract balance as an untainted environment value, and all control registers at ⊥.

Before processing a block, the analysis fingerprints the taint state and skips blocks already visited with the same fingerprint (Lines 5–6). The fingerprint hashes the bounded stack slice, the *overflow* bit, and each tracked taint tuple (*src*, *tainted*, *checked*, *origins*), preserving checked-state and provenance distinctions. Before pruning a repeated state, the implementation confirms structural equality; no fingerprint collision was observed in our evaluation. Each instruction is processed by its transfer function TaintStep (Line 9). At call/jump sites, the cross-continuation taint bridge (Sect. 3.5.3) propagates taint through the target's savelist summary (Line 11). A finding is recorded when a sink operand is tainted and unchecked (Line 13). When a successor edge leads back to an already-visited block (a back edge, indicating a loop) and the unroll bound MaxUnroll is exceeded, a conservative loop summary reports any tainted values reaching sinks within the loop body without further

unrolling (Line 16). The priority function favors blocks containing sensitive operations and those whose stack carries unguarded tainted values, directing exploration toward the most vulnerability-prone paths first.

Termination. The analysis terminates because the state space is finite. Each stack/register slot ranges over a finite lattice, and only the top K_s stack entries are tracked. Combined with the finite number of basic blocks and the loop bound MaxUnroll, the visited set $\mathcal{V}$ is finite and the worklist eventually empties.

3.6 Security Detectors

TasmScan implements detectors for 5 of the 8 defect classes defined by TONScanner [26], operating on TASIR semantic labels and savelist-enhanced taint facts. Each detector is described by a *vulnerability definition* followed by its *detection rule*. The rules are written in terms of TASIR-level and helper predicates rather than concrete opcode patterns or other low-level matching heuristics.

V1 — Unhandled Bounced Message. *Definition.* In TON's asynchronous messaging, a bounced message re-enters `recv_internal`. If the contract does not check the bounced flag, it processes the bounce as a legitimate message, causing permanent fund loss.
Rule. The detector searches an entry prefix of `recv_internal` for bounced-flag extraction sites that are followed by a guard. The implementation recognizes several equivalent bit-extraction idioms. Let $\mathit{Prefix}(f)$ denote the scanned entry prefix of function f, $\mathit{Bounce}(i)$ hold when instruction i extracts the bounced flag, and $G(i)$ hold when instruction i is followed by a guarding conditional branch or throw. Formally:

$$\nexists\, i \in \mathit{Prefix}(\texttt{recv\_internal}) :\ \mathit{Bounce}(i) \wedge G(i)$$

V2 — Bad Randomness. *Definition.* Predictable blockchain values (`NOW`, `BLOCKLT`, `LTIME`) used as random seeds allow miners or validators to predict outcomes, enabling front-running or fund manipulation.
Rule. The detector collects randomness-related instructions and security-sensitive sinks within each function. A finding is reported when a randomness-to-sensitive pair is found in program order, using this ordering as a heuristic proxy that predictable values may affect a critical action. Let $\mathit{Rand}(f)$ and $\mathit{Sens}(f)$ denote the randomness-related sites and security-sensitive sinks in function f, respectively, and let $\mathit{Before}(i, j)$ denote program order within a function. Formally:

$$\exists\, f \in F,\ r \in \mathit{Rand}(f),\ s \in \mathit{Sens}(f) :\ \mathit{Before}(r, s)$$

V3 — Lack of end_parse. *Definition.* A `Slice` created from a `Cell` via `CTOS` is parsed but never validated with `ENDS`, potentially accepting malformed data with hidden payloads.
Rule. For each function, the detector identifies slices that are created from cells and subsequently read, and compares them against `ENDS` validation sites. A finding is reported when at least one parsed slice lacks a corresponding end-of-slice validation. Let $\mathit{Parsed}(f)$ denote the parsed slices in function f, and let $\mathit{Ended}(p, f)$ hold when parsed slice p is matched by an `ENDS` validation in function f. Formally:

$$\exists\, f \in F,\ p \in \mathit{Parsed}(f) :\ \neg \mathit{Ended}(p, f)$$

V4 — Precision Loss. *Definition.* An integer division followed by multiplication silently truncates the intermediate quotient; the fused `MULDIV` opcode avoids this by computing the full-width product before dividing.
Rule. The detector looks for a nearby multiplication that follows a division. If the division site is not a fused `MULDIV`-family instruction, written as $\neg \mathit{Fused}(d)$, the pair is flagged as potential precision loss. Let $\mathit{Div}(f)$ and $\mathit{Mul}(f)$ denote the division and multiplication sites in function f, and let $\mathit{Near}(i, j)$ hold when the two sites are within the detector's short matching range. Formally:

$$\exists\, f \in F,\ d \in \mathit{Div}(f),\ m \in \mathit{Mul}(f) :\ \mathit{Before}(d, m) \wedge \mathit{Near}(d, m) \wedge \neg \mathit{Fused}(d)$$

V5 — Improper Modifier. *Definition.* A function lacking the `impure` modifier that directly or transitively throws, sends messages, modifies storage, or writes globals is side-effecting. A call to such a function whose return value is unused indicates a missing `impure` annotation.
Rule. Let $\mathit{Call}(f)$ be the call sites in function f, $\mathit{target}(i)$ the resolved callee of call i, $\mathit{NoImpure}(g)$ hold when g lacks `impure`, $\mathit{SE}(g)$ when g is side-effecting, and $\mathit{DropRet}(i, g)$ when the return value of i is unused or g returns no value. Formally:

$$\exists\, f \in F,\ i \in \mathit{Call}(f),\ g{=}\mathit{target}(i) :\ \mathit{NoImpure}(g) \wedge \mathit{SE}(g) \wedge\ \mathit{DropRet}(i,\ g)$$

4 Evaluation

We evaluate TasmScan with five research questions:

- **RQ1:** How accurate is continuation-target resolution?
- **RQ2:** How does TasmScan compare with TSA under identical runtime budgets?
- **RQ3:** How effective is TasmScan on five defect classes?
- **RQ4:** How much does path sensitivity contribute to detection?
- **RQ5:** What is the defect distribution and precision on Registry?

4.1 Experimental Setup

Implementation and Environment. TasmScan is implemented in approximately 17,000 lines of Python code (excluding comments, blank lines, and docstrings), organized into five modules mirroring the pipeline (Fig. 2). BOC deserialization uses `pytoniq-core` [39], and the opcode database is derived from the official TVM specification [23]; all other components are developed from scratch. The loop-unrolling bound MaxUnroll is set to 3, and the taint stack-tracking depth K_s is 64. All experiments run on Apple M4 Max with 16 cores, 128 GB RAM, and macOS 26.3. TONScanner is distributed as a pre-built Linux x86-64 ELF binary without source code; the RQ5 comparison therefore runs inside a Docker container (Ubuntu 20.04, x86-64 emulation via Rosetta 2).

Datasets. Two datasets are used: **Registry**, a set of 2,921 unique TON contracts collected by us from the TON verifier registry [30] (mainnet and testnet, February 2026), for RQ1, RQ2, and RQ5. Registry is the deduplicated full snapshot; we apply no outcome-based filtering. **Benchmark** contains 208 contracts with published manual labels from TONScanner [26], for RQ3 and RQ4. RQ3–RQ5

Table 4: Resolution results. All tiers have 100% target precision and exact-set agreement; exact means no missing or spurious oracle targets.

Edge Tier	Edges	Ratio	Oracle
Statically determined	284,066	96.4%	Structural
Solver-resolved	9,452	3.2%	Dictionary
Savelist-dependent	1,028	0.4%	Source audit
Total	**294,546**	**100%**	

Table 5: Cumulative ablation of continuation-resolution techniques.

Configuration	Unresolved	Resolved
Baseline (intracont. sim. + known-prefix)	11,081	—
+ intercont. propagation	10,612	469
+ reachability pruning	6,980	3,632
+ *savelist* propagation	**5,952**	**1,028**

metrics are computed at the *contract–class pair* level. The Benchmark contains 190 pairs labeled as vulnerable and 6 pairs labeled as non-vulnerable by TONScanner across the five evaluated classes.

For baselines, RQ2 compares TasmScan with TSA [13]; RQ3 and RQ5 compare it with TONScanner [26]. RQ4 evaluates one component ablation, TasmScan-NoPS (path sensitivity disabled). All tools use bytecode-only BOC input and identical hardware. Per-contract timeouts are 1,000 s for RQ2 and 30 s for RQ4. TSA is run with its default settings.

4.2 RQ1: Continuation Resolution Completeness and Accuracy

RQ1 evaluates target resolution along two dimensions. *Accuracy* asks whether resolved targets are correct; *completeness* asks how many edges each technique resolves. Since all 2,921 contracts originate from the TON verifier registry with verified source code, we construct a deterministic verification oracle for each resolved edge.

Oracle construction. We assign each dynamic edge to one of three tiers based on its resolution mechanism (Table 4). **Statically determined** edges (284,066, 96.4%) are conditional branches such as `IFELSE`, `IF`, `IFNOT`, and `IFJMP` whose targets are the `PUSHCONT` operands immediately preceding the branch instruction. Ground truth is extracted structurally from the bytecode: the target continuation identifiers are uniquely determined by instruction position, making verification fully automatic and provably correct. **Solver-resolved** edges (9,452, 3.2%) are dictionary-dispatch edges (`CALLDICT`, `DICTIGETJMPZ`) resolved by reading compile-time dictionary content from `c3`. Ground truth is the set of dictionary entries; we cross-verify these against source-level function definitions, confirming that the method identifiers match the declared functions. **Savelist-dependent** edges (1,028, 0.4%) are indirect register calls (`EXECUTE`/ `JMPX` via `PUSHCTR`) whose targets depend on savelist propagation. These edges are verified by source-code audit of the 129 affected contracts: for each edge, we trace the register-flow chain through the source to confirm that the resolved target matches the call site semantics. Of the 1,028 edges, 1,021 (99.3%) follow the `PUSHCTR c3` → `EXECUTE` pattern; the remaining 7 edges across 4 contracts are resolved via alternative solver strategies (*e.g.*, dictionary dispatch). All 1,028 edges are confirmed correct.

Accuracy result. For each edge we compare TasmScan's resolved target set against the oracle-verified ground truth. Table 4 shows that all 294,546 edges achieve **100% precision**: the resolved targets match the ground truth exactly on every edge, verified per-tier as described above. Per-contract audit reports for the savelist tier are provided in the artifact package.

Completeness result. Table 5 shows the cumulative contribution of each resolution technique, measured by the number of unresolved indirect call targets (i.e., `EXECUTE`/`JMPX` edges with unknown destinations). The baseline leaves 11,081 targets unresolved. Inter-continuation propagation resolves 469 targets by propagating entry-shape information across continuation boundaries. Reachability pruning eliminates 3,632 targets that originate from unreachable code—dead continuations extracted from data-cell references (*e.g.*, `PUSHREFSLICE`) rather than executable control flow. Finally, *savelist* propagation resolves 1,028 targets across 129 contracts (4.4% of the corpus) by tracking the `PUSHCTR c0/c3`→`EXECUTE` pattern at the IR level. After all four techniques, 5,952 indirect calls remain unresolved—these arise from continuations passed through complex stack manipulations where the analysis cannot trace the target origin—and are conservatively left as unknown in the call graph. All 294,546 resolved targets achieve **100% precision** (Table 4), i.e., every resolved target is verified correct.

Table 6: Budgeted exploration on Registry; times are p50/p95.

Tool	Outcomes	Time (s)
TasmScan	2,921 success; 0 timeout; 0 crash	0.24/5.52
TSA	2,834 success; 12 timeout; 75 crash	4.10/64.90

4.3 RQ2: Budgeted Exploration vs TSA

RQ2 evaluates whether TasmScan can analyze the entire Registry reliably and how its throughput compares with TSA.

Result. TasmScan achieves a **100% success rate** with zero crashes and zero timeouts, while TSA succeeds on 97.0% of contracts with 12 timeouts and 75 crashes (Table 6). TSA's 1,000 s limit is its default budget; TasmScan's maximum observed runtime is 56.6 s, so this budget does not truncate its runs. TasmScan's median analysis time (0.24 s) is **17× faster** than TSA's (4.10 s), and the 95th-percentile gap widens to 12× (5.52 s vs. 64.90 s). This speed advantage stems from TasmScan's dataflow-analysis approach, which avoids the path explosion inherent in TSA's symbolic execution.

Code coverage. Beyond throughput, the two tools differ fundamentally in analysis scope. TasmScan computes one whole-CFG dataflow fixed point rather than making one literal pass: instructions are revisited until states stabilize, with loop edges bounded to three unrollings. It reached every reachable instruction on all 2,921 contracts within the 1,000 s budget. TSA, as a symbolic executor, explores concrete paths one at a time; its internal coverage tracker

Table 7: Per-class detection rate on Benchmark.

Defect Class	Labeled	Detected	Det. Rate
Bad Randomness	2	2	100.0%
Improper Modifier	6	6	100.0%
Unhandled Bounced Msg	56	56	100.0%
Lack of end_parse	82	82	100.0%
Precision Loss	44	35	79.5%
Overall	**190**	**181**	**95.3%**

Table 8: Contract-level precision comparison.

	TasmScan		TONScanner	
Language	**TP/FP**	**Precision**	**TP/FP**	**Precision**
FunC	138 / 6	95.8%	144 / 6	96.0%
Tact	43 / 0	100%	46 / 0	100%
Combined	**181 / 6**	**96.8%**	**190 / 6**	**96.9%**

reports *transitive instruction coverage*—the fraction of reachable instructions visited across all explored paths—which is inherently bounded by the exploration budget and subject to path explosion. On the 87 contracts (3.0%) where TSA crashes or times out, it produces no analysis results at all, whereas TasmScan completes with full coverage. This difference has a direct downstream impact: a defect detector can only flag patterns in code it has analyzed, so TasmScan's complete coverage eliminates an entire class of false negatives that arise from incomplete exploration.

4.4 RQ3: Detection Effectiveness

RQ3 evaluates TasmScan on five defect classes from **Benchmark**. Three classes are excluded: Global Variable Redefined and Inconsistent Data require source-level analysis (variable re-declaration and per-function cell-layout comparison), and Unchecked Return requires source-level variable-name tracking—none of which can be reliably performed at the bytecode level. For each class, we measure *detection rate*: the fraction of contracts labeled as vulnerable that TasmScan also flags. TSA does not provide high-level defect-class labels; its output reports low-level execution errors (*e.g.*, `cell-underflow`) rather than vulnerability types, so it is not included in this comparison.

Results. TasmScan detects 181 of 190 vulnerable contract–class pairs (95.3%); all cases in four classes are detected, and the nine misses are Precision Loss (Table 7). It also reports findings for all six pairs labeled non-vulnerable, which are the false positives analyzed below. Table 8 shows 96.8% overall precision, close to TONScanner's 96.9%, without requiring source code.

False-positive analysis. Six contract–class pairs are over-reported: 4 from Unhandled Bounced Message and 2 from Lack of end_parse. The former handle the bounced flag beyond the detector's entry-prefix window; the latter use parsing patterns not captured by its opcode heuristic.

Table 9: Per-class findings and precision on Registry.

Defect Class	Affected (%)	Audit TP/FP	Precision
Lack of end_parse	2,795 (95.7)	87/6	93.5%
Unhandled Bounce	1,790 (61.3)	77/15	83.7%
Bad Randomness	304 (10.4)	68/6	91.9%
Improper Modifier	54 (1.8)	29/6	82.9%
Precision Loss	854 (29.2)	53/19	73.6%
Overall	**2,806 (96.1)**	**314/52**	**85.8%**

False-negative analysis. All nine misses are Precision Loss. Recompilation and rescanning confirm that chained arithmetic separates the vulnerable division and later multiplication beyond the 12-instruction matching window. This is a rule limitation rather than a bytecode-availability issue; tracking division results by taint would remove the fixed-window constraint.

4.5 RQ4: Path-Sensitivity Ablation

Under a 30 s timeout, the full configuration detects 92.6% (five contracts time out), whereas path-insensitive TasmScan-NoPS detects 91.1% (four time out). The remaining 1.5-point gap reflects precision lost by merging path-disjoint states. Separately, *savelist* propagation resolves 1,028 additional targets across 129 contracts (Sect. 4.2). Disabling it does not change the current detection totals because the five detectors ultimately match local opcode patterns; its detection-level contribution is therefore a zero-difference ablation, while its measurable contribution is CFG completeness and support for future cross-continuation detectors.

4.6 RQ5: Full-Scale Defect Analysis

RQ5 applies TasmScan to the full Registry to measure defect prevalence and estimate per-class precision via stratified sampling.

Defect Distribution. TasmScan flags defects in 2,806 of 2,921 Registry contracts (96.1%). Lack of end_parse is most prevalent (2,795; 95.7%), followed by Unhandled Bounced Message (1,790; 61.3%); Table 9 gives all classes.

Precision Sampling. We audit a stratified sample (95% confidence, 10% margin) against verified source via independent review by two authors (Table 9). Lack of end_parse is reported under the TONScanner convention that treats Tact-compiler-omitted `ENDS` as a true positive (compiler deficiency), yielding 93.5%. Unhandled Bounced Message reaches 83.7% after review of bounced handlers and send paths. Improper Modifier reaches 82.9%; its remaining false positives are context-dependent side effects. Precision Loss is least precise (73.6%) because its short-window rule lacks richer value-flow reasoning.

Comparison with TONScanner on Registry. We attempt TONScanner on 343 Registry contracts (366 contract–class pairs). Only 280 are analyzable: 147 are FunC and 133 Tact contracts can be recompiled; 50 Tact projects fail to compile and 13 inputs are non-FunC. Its parser also rejects some generated FunC, further limiting coverage. Across the 366 sampled contract–class pairs, TONScanner flags 32 findings with 84.4% precision (27 TP, 5 FP), compared with TasmScan's 85.8% precision (314 TP, 52 FP) on the same sample.

Thus, the main limitation is source-language coverage rather than flagged-finding precision.

5 Related Work

5.1 Smart Contract Analysis

Smart contract analysis is well explored for the EVM [5, 6, 9, 14, 18, 20, 21, 28, 32], including formal semantics [17, 25] and vulnerability taxonomies [4, 7]. For CFG construction from EVM bytecode, EtherSolve [10] and EVMLiSA [2, 3] use stack simulation to resolve EVM jumps. TasmScan adapts these stack-simulation ideas to TVM's continuation model and extends them with *savelist* abstraction, which has no analogue in EVM. Beyond Ethereum, WASAI [8], VETEOS [19], and MoveScan [27] target other blockchains; none handle first-class continuations or *savelist* semantics.

Within TON, TONScanner [26] defines eight defect classes and operates on the FunC compiler IR, providing detailed source-level analysis but requiring access to source code; as shown in our evaluation (Sect. 4.6), its FunC parser rejects a substantial fraction of our Registry sample due to language-version incompatibilities. Misti [22] similarly targets *Tact* source and is limited to contracts written in that language. TSA [13] operates at the bytecode level via symbolic execution, modeling *savelist* operations per path, but does not build a static summary of *savelist* effects across all paths—each path maintains a concrete *savelist* structure with no join or widening. Yanovich *et al.* [38] derive an audit checklist from professional TON audit reports, providing a complementary knowledge-driven perspective; BugMagnifier [37] complements static analysis with dynamic transaction simulation for runtime validation. TasmScan is positioned as a source-free bytecode analyzer with a dedicated *savelist* model, removing the source-code requirement of TONScanner/Misti, replacing per-path execution of TSA with a static dataflow model, and handling first-class continuation semantics natively.

5.2 Continuation Semantics

Continuations have a long history in programming-language theory [1, 15], including continuation-sensitive CFA [33, 34] and pushdown analyses [16, 35] for first-class control. These techniques target languages where continuations capture lexical environments; the analysis challenge is representing the unbounded set of captured bindings. TVM continuations differ fundamentally: they carry explicit register snapshots (*savelists*) rather than captured environments, and invocation restores those registers automatically. The analysis challenge is therefore to statically approximate which registers are saved at each call site and what values they hold—a problem closer to interprocedural register analysis than to environment abstraction. The dedicated savelist model in Sect. 3.4 addresses this challenge. To our knowledge, TasmScan provides the first static over-approximation of TVM *savelist* behavior.

Unlike fixed interprocedural call/return pairs, a first-class continuation targets the value held in a register and restores an explicit savelist rather than an implicit frame; analysis must recover both.

6 Discussion

Limitations. For *bounded analysis*, the soundness guarantee of Thm. 1 applies to the *savelist* model rather than the full analysis chain, so the analysis may miss vulnerabilities due to bounded loop unrolling, path limits, and 5,952 indirect calls whose continuation targets remain unresolved after all resolution techniques (Sect. 4.2). Such calls remain unknown edges and preserve conservative taint at the transfer, but their destinations cannot be enumerated; they do not affect the current local detectors but may hide paths from future cross-continuation detectors. The taint-state cache uses structural equality after hashing, preventing hash collisions from merging unequal states; none occurred in our evaluation. For *coverage scope*, TasmScan targets 5 of 8 TONScanner defect classes; the remaining three require source-level semantics erased during compilation.

Generality and Future Work. The savelist abstraction is TVM-specific, but data-driven target resolution and TASIR's typed CFG generalize to other indirect-jump bytecodes; EVM itself has no savelists. IFDS would neither remove our bounded operand-stack domain nor support the non-distributive, path-specific guard state. Future work will add savelist-aware detectors and solver-assisted indirect-call resolution.

Threats to Validity. For *internal* validity, ground-truth labels are cross-validated against TONScanner's published annotations; the RQ1 oracle is exact by construction for 96.4% of edges and empirically verified for the remainder. For *external* validity, our dataset contains open-source TON contracts only and may not cover all deployment settings. For *construct* validity, detection-rate and precision estimates are bounded by TONScanner's label space; manual auditing on a stratified sample provides cross-validation but cannot fully eliminate labeling bias.

7 Conclusion

This paper presented TasmScan, a source-free static analysis framework for TVM bytecode whose key contribution is a static dataflow model for *savelist* semantics, enabling taint propagation across continuation boundaries. Compared with TSA, the only prior bytecode-level analyzer for TVM, TasmScan analyzes all contracts without crashes or timeouts, runs an order of magnitude faster, and covers every reachable instruction by construction. Evaluation on 2,921 real-world contracts confirms high detection rates and practical precision across five defect classes, matching the source-level TONScanner without requiring access to source code.

Data Availability Statement

Our replication package is available online at https://github.com/yxsec/TasmScan_artifact.

Acknowledgments

This work was supported by the Singapore Ministry of Education Academic Research Fund Tier 2 (T2EP20224-0003) and Tier 1 (RG12/23), and the Nanyang Technological University Centre for Computational Technologies in Finance (NTU-CCTF). The views expressed are those of the authors and not necessarily those of MOE or NTU-CCTF.

References

[1] Andrew W Appel. 1992. *Compiling with Continuations.* Cambridge University Press. doi:10.1017/CBO9780511609619

[2] Vincenzo Arceri, Saverio Mattia Merenda, Greta Dolcetti, Luca Negrini, Luca Olivieri, and Enea Zaffanella. 2024. Towards a Sound Construction of EVM Bytecode Control-Flow Graphs. In *Proceedings of the 26th ACM SIGPLAN International Workshop on Formal Techniques for Java-like Programs (FTfJP).* ACM. doi:10.1145/3678721.3686227

[3] Vincenzo Arceri, Saverio Mattia Merenda, Luca Negrini, Luca Olivieri, and Enea Zaffanella. 2025. EVMLiSA: Sound Static Control-Flow Graph Construction for Ethereum Bytecode. *Blockchain: Research and Applications* (2025). doi:10.1016/j.bcra.2025.100384

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). *Proceedings of the 6th International Conference on Principles of Security and Trust (POST)* (2017), 164–186. doi:10.1007/978-3-662-54455-6_8

[5] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P).* IEEE, 161–178. doi:10.1109/SP46214.2022.9833721

[6] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 454–469. doi:10.1145/3385412.3385990

[7] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022), 327–345. doi:10.1109/TSE.2020.2989002

[8] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. 2022. WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* ACM, 703–715. doi:10.1145/3533767.3534218

[9] ConsenSys Diligence. Accessed: March 2026. Mythril. https://github.com/ConsenSysDiligence/mythril.

[10] Federico Contro, Mirco Crosara, Mariano Ceccato, and Mila Dalla Preda. 2021. EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode. In *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC).* IEEE, 127–137. doi:10.1109/ICPC52881.2021.00021

[11] DefiLlama. Accessed: March 2026. TON DeFi Dashboard — Total Value Locked. https://defillama.com/chain/TON.

[12] Nikolai Durov. Accessed: March 2026. The Open Network. https://ton.org/whitepaper.pdf.

[13] Esprito Tech. Accessed: March 2026. TSA: TON Symbolic Analyzer. https://github.com/espritoxyz/tsa.

[14] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *Proceedings of the 2nd IEEE International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB).* IEEE, 8–15. doi:10.1109/WETSEB.2019.00008

[15] Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 180–190. doi:10.1145/73560.73576

[16] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown Control-Flow Analysis for Free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 691–704. doi:10.1145/2837614.2837631

[17] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *Proceedings of the 2018 IEEE 31st Computer Security Foundations Symposium (CSF).* IEEE, 204–217. doi:10.1109/CSF.2018.00022

[18] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE).* ACM, 259–269. doi:10.1145/3238147.3238177

[19] Levi Taiji Li, Ningyu He, Haoyu Wang, and Mu Zhang. 2024. VETEOS: Statically Vetting EOSIO Contracts for the "Groundhog Day" Vulnerabilities. In *31st Annual Network and Distributed System Security Symposium (NDSS).* doi:10.14722/ndss.2024.24972

[20] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS).* ACM, 254–269. doi:10.1145/2976749.2978309

[21] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE).* ACM, 778–788. doi:10.1145/3377811.3380334

[22] Nowarp. Accessed: March 2026. Misti: Static Analysis Tool for Tact Smart Contracts. https://github.com/nowarp/misti.

[23] The Open Network TON Organization. Accessed: March 2026. TVM Instructions Specification. https://github.com/ton-org/tvm-spec.

[24] Daniel Perez and Benjamin Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *Proceedings of the 30th USENIX Security Symposium.* 1325–1341.

[25] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS).* ACM, 621–640. doi:10.1145/3372297.3417250

[26] Hao Song, Teng Li, Jiachi Chen, Ting Chen, Beibei Li, Zhangyan Lin, Yi Lu, Pan Li, and Xihan Zhou. 2025. Enhancing The Open Network: Definition and Automated Detection of Smart Contract Defects. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE).* IEEE, 1281–1293. doi:10.1109/ICSE55347.2025.00119

[27] Shuwei Song, Jiachi Chen, Ting Chen, Xiapu Luo, Teng Li, Wenwu Yang, Leqing Wang, Weijie Zhang, Feng Luo, Zheyuan He, Yi Lu, and Pan Li. 2024. MoveScan: Smart Contract Security Analysis. In *Proceedings of ISSTA 2024.* ACM, 1682–1694. doi:10.1145/3650212.3680391

[28] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the 46th International Conference on Software Engineering (ICSE).* ACM, 1–13. doi:10.1145/3597503.3639117

[29] TON Foundation. Accessed: March 2026. TON Blockchain. https://docs.ton.org/.

[30] TON Foundation. Accessed: March 2026. TON Verifier — Smart Contract Source Registry. https://verifier.ton.org/.

[31] Tonscan. Accessed: March 2026. TON Blockchain Statistics. https://tonscan.org/stats.

[32] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS).* ACM, 67–82. doi:10.1145/3243734.3243780

[33] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP).* ACM, 51–62. doi:10.1145/1863543.1863553

[34] Dimitrios Vardoulakis and Olin Shivers. 2011. CFA2: A Context-Free Approach to Control-Flow Analysis. *Logical Methods in Computer Science* 7, 2 (2011). doi:10.2168/LMCS-7(2:3)2011

[35] Dimitrios Vardoulakis and Olin Shivers. 2011. Pushdown Flow Analysis of First-Class Control. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP).* ACM, 159–170. doi:10.1145/2034773.2034785

[36] Gavin Wood. Accessed: March 2026. Ethereum Yellow Paper. https://ethereum.github.io/yellowpaper/paper.pdf.

[37] Yury Yanovich, Victoria Kovalevskaya, Maksim Egorov, Elizaveta Smirnova, Matvey Mishuris, Yash Madhwal, Kirill Ziborov, Vladimir Gorgadze, and Subodh Sharma. 2025. BugMagnifier: TON Transaction Simulator. *CoRR* abs/2509.24444 (2025). doi:10.48550/arXiv.2509.24444

[38] Yury Yanovich, Sergey Sobolev, Yash Madhwal, Kirill Ziborov, Vladimir Gorgadze, Victoria Kovalevskay, Elizaveta Smirnova, Matvey Mishuris, and Subodh Sharma. 2025. From Paradigm Shift to Audit Rift: Empirical Analysis and Validation of Security Audit Methodologies for Asynchronous Smart Contract Systems. *CoRR* abs/2509.10823 (2025). doi:10.48550/arXiv.2509.10823

[39] yungwine. Accessed: March 2026. pytoniq-core: TON Blockchain Data Structures for Python. https://github.com/yungwine/pytoniq-core.

Received 2026-03-26; accepted 2026-06-18