Recover Function Signature from Combined Constraints

CCS 2025

Haohui Huang, Yue Liu, Yuxi Cheng, Haiyang Wei, Jiamu Liu, Yu Wang, Linzhang Wang
(Nanjing University, Southeast University, QianXin Group)

Why Function Signature Recovery Matters

Function signatures underpin binary analysis

- Reconstructing program semantics from stripped binaries starts from the signature
- Downstream uses named in the paper: code summarization, vulnerability discovery, control-flow integrity, code similarity (diaphora)

What makes it hard

- Aggressive compiler optimization removes much of the information the recovery needs
- Existing methods lose scalability and accuracy across diverse architectures and optimization levels

Two sub-problems

- Identification of parameters and return values
- Recovery of parameter types

Limitations of Existing Approaches

Parameter / return detection

- Traditional methods use live variable analysis and def-use analysis to track register state
- Variadic identification relies on pattern matching, which compiler optimization breaks
- Focusing only on argument-passing registers fails for functions with more than six parameters
- AI-based methods need architecture-specific multitask models; **minor assembly changes from different optimizations shift predictions significantly**

Parameter type recovery

- Rule-based: dynamic analysis for type propagation (REWARDS) or probabilistic inference (OSPREY)
- Degrade because constraints are solved at the assembly level, which is inefficient
- Commercial and open tools (IDA, Ghidra, Binary Ninja, angr) rely on extensive heuristic rules

Background: Factor Graph and Belief Propagation

Factor graph

- A bipartite graph G(V, F, E): variable nodes V and factor nodes F, an edge denoting a dependency
- The whole graph denotes the joint distribution of all variable nodes
- Only ratios between competing assignments matter; a factor's absolute magnitude does not
- In CDA, factor nodes are the decompilers' outputs and the correction rules

Belief propagation

- Message-passing algorithm that efficiently computes marginal distributions of variable nodes
- Variables and factors iteratively exchange messages and beliefs combine all incoming factor messages
- Used here to compute the posterior of the variables of interest — whether a function has a return value, how many parameters it has

Motivating Example: customPrintf (x64, O2)

```
void doOutput(OutputContext * context_ptr,short max,bool should{
    customPrintf(context_ptr->ptr,should,max, "This is a %hd
        message with a maximum length.", max);
}                                                  Pair Callsite Hint

void customPrintf(OutputContext *context, bool shouldOutput,
                  short max,const char *format, ...) {
    va_list args;
    va_start(args, format);
    if (shouldOutput && context->maxLength < max - 1){
        if(context->ptr != NULL){
            vsnprintf(context->ptr, context->maxLength+1,
                format, args)}};
    }                          Operator/Operad Hint
    va_end(args);
}
```

Existing Decompilers' Outputs

```
int sub_11F0(__int64 a1,int a2,__int16 a3,const char *
a4, ...)
```

```
void FUN_001011f0(//omit,long param_9,int param_10,
short param_11,char* param_12,//omit,undefined8 param_14)
```

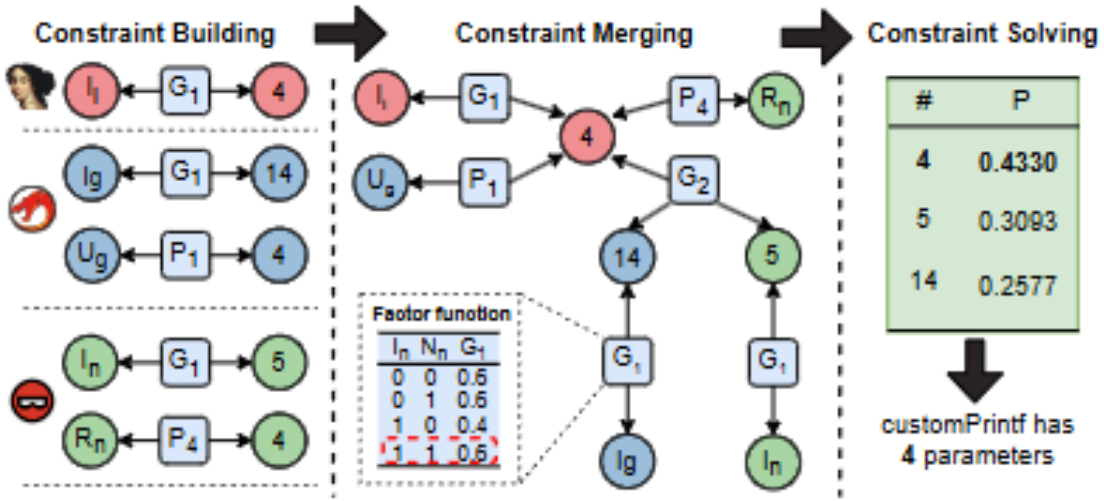
Unused ParametersHint

```
int32_t sub_4011f0(void *arg1,int32_t arg2,int16_t arg3,
char *arg4, int32_t arg5 @rax)
```

Decompiation Error Hint

CDA Outputs

```
int fun_11F0(struct * a1, bool a2, short a3, char *a4,
...)
```



Strengths of each decompilers

- IDA: parameter identification (variadic template)
- Ghidra: return type identification
- Ninja: parameter type recovery

Key Observations

Observation 1. Patterns in decompiled code can correct decompiler errors

- A signature with 14 parameters is rare in practice, and param_2–param_8 are never used in the body — the count has been over-estimated
- A parameter bound to a register outside the calling convention (@rax) is a hint about parameter detection, not a real parameter
- A non-pointer-sized variable that is only ever compared to 0 in the whole binary has a high probability of being a bool
- Field accesses such as context->ptr appear as base-offset patterns (a1 + 0x8) in decompiled output, implying a pointer type
- Format specifiers at the call site reinforce the type determination of the matching argument

Observation 2. Each decompiler outperforms the others in specific aspects

- IDA is particularly effective at detecting variadic functions — the only tool that identifies customPrintf as variadic
- Pointer recovery is a special case: decompilers are conservative, so a variable is typed as a pointer only when they are confident
- Therefore if a single decompiler calls a variable a pointer, the probability of it being a primitive type elsewhere should be treated as negligible

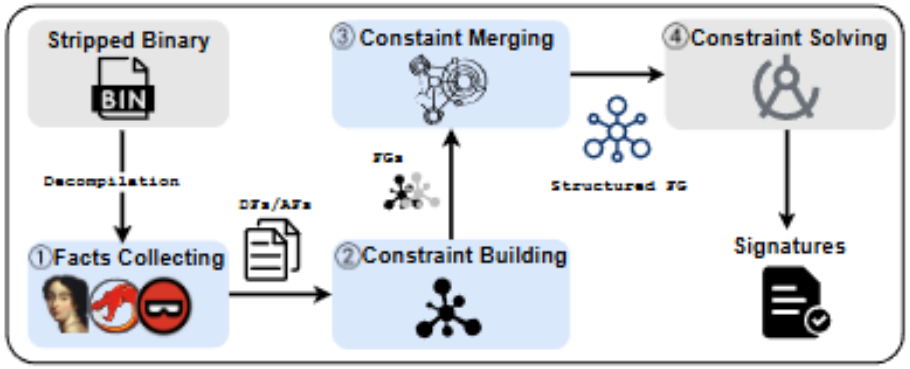


Figure 4: Workflow of CDA.

Facts Collecting

Parameter/Return related Decompilation Facts

D_1	Register(d, f, v, r)	:	In decompiler d, variable v in function f has register r
D_2	Signature$(d, f, ret, Params)$	:	In decompiler d, function f has return ret and parameters $Parames$
D_3	CallSiteSign$(d, f_1, f_2, cs, ret_{cs}, Args_{cs})$	:	function f_1 invoke function f_2 at call site cs where output is ret_{cs} and arguments set is $Args_{cs}$
D_4	FormatStr$(d, f_1, f_2, cs, i, str)$	:	function f_1 invoke function f_2 at call site cs where the i_{th} argument is a formatted string str
D_5	Variadic(d, f, i)	:	In decompiler d, function f is identified as variadic function with variadic position i

Type related Decompilation Facts

D_6	BinStmt(d, v_1, v_2, o)	:	In a binary expression, the operator between variables v_1 and v_2 is o.
D_7	AssignStmt(d, v_1, v_2)	:	An assignment statement of the form $v_1 = v_2$
D_8	BaseOffset(d, v_i, v_j)	:	The code form of base address plus offset, for example,$*(v_1 + v_2)$
D_9	SwitchStmt(d, v, C)	:	The variable v appears in a switch statement, with the cases being a set of constants C

Analysis Facts

A_1	DirectRet$(d, f, DRets)$	:	In decompiler d,the set of parameters directly returned by function f
A_2	DecIntroVars$(d, f, DIVs)$	:	In function f, the unconventional variables $DIVs$ introduced by tool d
A_3	ParamUnused(d, f, UUs)	:	In decompiler d, the set UUs represents the unused parameters of function f
A_4	Definition$(d, f, cs, arg, Defs)$	:	The set of statements defining the actual argument arg that appear before the call site cs when function f passes arg.

Helper Functions

H_1	Type(d, v)	:	Return type of variable v in decompiler d
H_2	Width(d, v)	:	Return width of variable v in decompiler d
H_3	CallConRegisters$(d, f, Regs)$	:	The register set $Regs$ corresponding to the calling convention of function f
H_4	Index(e, S, i)	:	The index i of element e in set S
H_5	Pair$(str, str_1 : arg_1, \ldots, str_m, arg_m)$	:	Match the format specifier str_i in the format string str with its corresponding argument arg_i

Figure 5: Facts Collected from Decompiler.

Constraint Rules I: Variadic / Parameter / Return

Category	ID	Condition	Constraint
general	G_1		$\text{Signature}(d, f, R_r, Params) \xrightarrow{p_d} \lvert\text{InferParam}(f)\rvert = \lvert Params\rvert \wedge \lvert\text{InferRet}(f)\rvert = T_r$ $\text{Variadic(d,f,i)} \xrightarrow{p_d} \text{InferVar}(f, i)$
	G_2	$rv_i, rv_j \in \text{FactorGraph}(d, f) \wedge rv_i \neq rv_j$	$rv_i \xleftrightarrow{p\downarrow} rv_j$
variadic	V_1	$p_i \in Params \wedge \text{Register}(d, f, p_i, r) \wedge r \notin Regs$	$\text{Register}(d, f, p_i, r) \xrightarrow{P_d} \text{InferVar}(f, min(Index(p_i, Params)))$
	V_2	$\lvert Args_{cs_i}\rvert \neq \lvert Args_{cs_j}\rvert \wedge \text{FormatStr}(d, \_, f, \_, idx, str)$	$\text{FormatStr}(d, \_, f, \_, idx, str) \xrightarrow{P_d} \text{InferVar}(f, idx + 1)$
	V_3	$\lvert\text{Params}\rvert > \gamma_p \vee \text{FormatStr}(d, \_, f, \_, idx, str)$	

General (G)

- G_1 has no condition — it always fires, and it is the only place where tool trust enters the model.
- G_2 adds mutual exclusion between variable nodes, because a function has exactly one arity — and as a side effect it keeps the graph loop-free.

Variadic (V)

- V_1 — a parameter bound to a register the ABI never uses for arguments means the function is probably variadic, starting at that position.
- V_2 — call sites passing different numbers of arguments, with a format string among them, means variadic at the format string's index plus one.
- V_3 — an absurd parameter count (over $\gamma\_p = 10$) or a format string means variadic; this one targets Ghidra, which counts `xmm0–xmm7` as parameters.

Constraint Rules I: Variadic / Parameter / Return

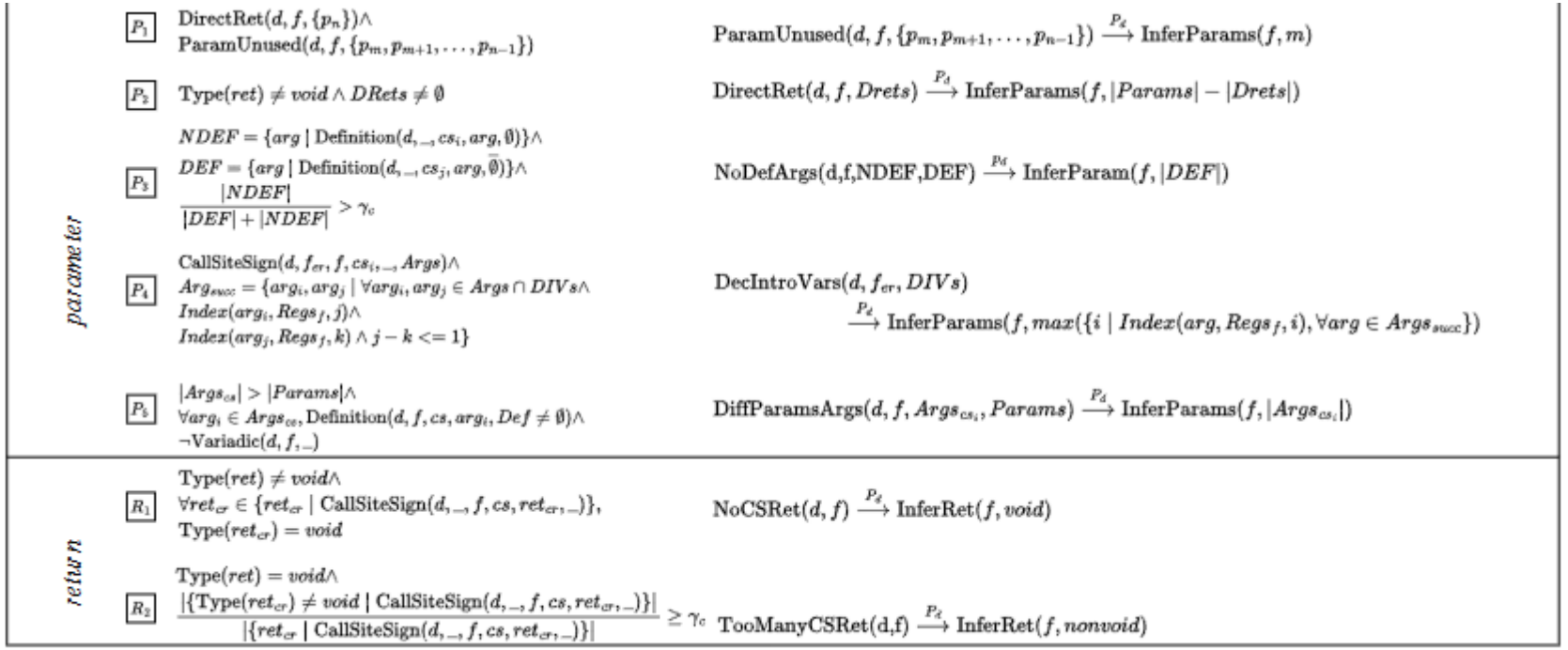


Figure 6: Constraints for Variadic/Parameter/Return Detection.

Parameter (P)

- `P1` — if a parameter is returned directly and only the first `m` are used, the real arity is `m` and the tail is decompiler noise.
- `P2` — when parameters are returned directly, the return is probably `void` and the real arity drops by the number of directly returned parameters.
- `P3` — if too many actual arguments have no defining statement, trust the count of defined arguments instead.
- `P4` — when a decompiler invents locals mapped onto argument registers, the real arity is the highest index among them, provided they are consecutive in the convention.
- `P5` — if every call site passes more fully-defined arguments than declared and nothing flagged it variadic, trust the call site; applied only to IDA and Ninja because Ghidra's argument identification is too weak.

Return (R)

- `R1` — a declared return value that no call site ever consumes is really `void`.
- `R2` — a nominally `void` function whose return is used at a quarter or more of its call sites probably does return something.

Constraint Rules II: Type Recovery

ID	Condition	Constraint
T_1		$\text{Type}(p_i) \xrightarrow{P_d} \text{Basic}(p_i)/\text{BasicPointer}(p_i)/\text{StructPointer}(p_i)/\text{VoidPointer}(p_i)$
T_2		$\text{AssignStmt}(d, v_1, v_2) \xrightarrow{P_d} \text{PropConstraint}(v_1, v_2)$
T_3	T_{3_a} $\text{BaseOffset}(d, v_i, 0)$	$\text{BaseOffset}(d, v_i, v_j) \xrightarrow{P_d} \text{BasicPointer}(v_i)$
	T_{3_b} $\text{BaseOffset}(d, v_i, v_j) \wedge v_j \neq 0$	$\text{BaseOffset}(d, v_i, v_j) \xrightarrow{P_d} \text{StructPointer}(v_i) \sqcap \text{Basic}(v_j)$
T_4	T_{4_a} $\text{BinStmt}(d, v_1, v_2, o) \wedge o \in \{==, \neq\} \wedge$ $Width(d, v_1) == PointerSize \wedge v_2 == 0$	$\text{BinStmt}(d, v_1, v_2, o) \xrightarrow{P_d} \text{Pointer}(v_1)$
	T_{4_b} $\text{BinStmt}(d, v_1, v_2, o) \wedge o \in \{==, \neq\} \wedge v_2 \neq 0$	$\text{BinStmt}(d, v_1, v_2, o) \xrightarrow{P_d} \text{Basic}(v_1) \wedge \text{Basic}(v_2)$
	T_{4_c} $\text{BinStmt}(d, v_1, v_2, o) \wedge o \notin \{==, \neq, +\}$	$\text{BinStmt}(d, v_1, v_2, o) \xrightarrow{P_d} \text{Basic}(v_1) \wedge \text{Basic}(v_2)$
T_5	$arg_i \in Args_{cs}, \forall i \in [0, min(\lvert Args \rvert, \lvert Params \rvert]$	$\text{CallSiteSign}(d, \_, f, cs, \_, Args_{cs}) \xrightarrow{p_d} \text{PropConstraint}(p_i, arg_i)$
T_6	$\wedge \exists k \in [0, min(Args_{cs_i}, Args_{cs_j})], \text{ConflictType}(a_{ik}, a_{jk})$	$\text{ConflictType}(a_{ik}, a_{jk}) \xrightarrow{p_d} \text{VoidPointer}(p)$
T_7		$\text{FormatStr}(d, \_, f, cs, idx, str) \wedge \text{Pair}(str, \{str_1 : arg_1, \ldots, str_m, arg_m\})) \xrightarrow{p_d} \text{EquType}(arg_i, str_i)$

Figure 7: Constraints for Type Recovery.

- T_1 — trust each decompiler's own type judgement with strength `P_d`; this is G_1 for types.
- T_2 — for $v_1 = v_2$, copy v_2's factor graph onto v_1; note that no data flow analysis happens, the graph is simply replicated.
- T_3a — a zero offset in a base-offset expression means a plain pointer.
- T_3b — a non-zero offset means a struct pointer plus a primitive offset, which is how `context->ptr` survives as `*(a1 + 0x8)`.
- T_4a — a null check at pointer width means a pointer.
- T_4b — comparison against a non-zero constant means both sides are primitives
- T_4c — bitwise or shift operators mean primitives, since you do not shift a pointer.
- T_5 — propagate the actual argument's graph onto the callee's formal parameter, which buys inter-procedural typing without doing data flow.
- T_6 — if argument types for one parameter slot cannot be reconciled across call sites, the parameter is probably `void *`.
- T_7 — match format specifiers one-to-one with the following arguments, so `%hd` pins `max` to `short`.

Merge Rules

- **Direct** — merge graphs unmodified. Used for parameter identification and return detection, where no decompiler consistently wins

- **Supersede** — if any tool asserts a property, keep it and discard conflicts. Applied to variadic functions (IDA) and pointer types, because decompilers only deny a property when no heuristic can confirm it

- **Downgrade** — down-weight rather than remove. Used for primitive types: if most hints say `bool`, competing primitives lose probability but stay in the graph

Implementation and Experiment Setup

Implementation

- Python, 5,500+ LOC, orchestrating parallel decompilation
- IDA Pro via idat CLI, Ghidra in headless mode, Binary Ninja
- Facts collection with Tree-sitter S-expression queries
- pgmpy for factor graph construction and belief propagation
- IDA 8.3, Ghidra 11.2.0, Binary Ninja 4.2.6455

Dataset

- OSPREY and GenNm datasets, de facto standards for binary analysis tools
- For GenNm, the top 200 most-starred GitHub projects recompiled with ghcc
- 5 architectures × 4 optimization levels → 43,944 binaries, 10M+ valid functions

Ground truth

- Traced from source code to compiled binary; trailing consecutive unused parameters in non-export functions are excluded from the count
- If a return value is never used anywhere in the binary, the ground truth is void
- Variable types come from DWARF debugging information

Evaluation: Variadic Function Detection

Table 2: Results for Variadic Function Detection.

Arch		x64				x86				AArch64				Arm				Mips				Avg.
opt		O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3	
IDA	precision	60.88	61.55	63.89	60.88	57.59	56.82	57.69	58.33	55.53	57.63	56.99	57.11	59.21	62.85	58.33	56.51	49.82	55.84	56.87	57.83	58.11
	recall	46.99	14.44	29.85	26.85	51.99	20.15	11.53	12.17	55.46	49.47	48.84	45.89	53.15	45.89	41.38	38.57	49.82	37.34	38.43	41.28	37.98
CDA	precision	63.97	64.44	55.78	55.02	56.18	58.88	59.85	58.63	55.06	53.88	58.34	52.23	56.18	65.44	58.28	57.17	51.82	52.49	54.39	57.47	57.28
	recall	64.7	64.7	62.22	64.13	56.89	50.34	49.45	53.05	56.76	51.12	62.03	51.19	54.33	46.12	54.29	52.46	49.82	47.34	50.66	53.37	54.75

Table 3: Results for Variadic Position Detection.

Arch	x64				x86				AArch64				Arm				Mips			
Opt	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3
IDA	64.40	55.40	94.89	95.45	97.73	58.03	59.53	57.25	91.27	97.35	97.13	94.58	96.15	96.15	96.15	95.05	96.10	96.10	96.41	96.41
CDA	67.94	99.86	96.62	95.42	97.73	97.73	97.41	97.25	93.31	98.31	94.05	95.73	96.15	96.15	92.29	92.29	96.10	96.10	82.68	96.41

Findings

- CDA achieves a significantly higher recall — an improvement of 16.77% — while keeping precision comparable to IDA
- Precision drops slightly, by 0.83%; manual analysis attributes this mainly to an overly low γ_p in rule V_3 and high priors assigned to certain tools
- A single set of priors was used across all compilation and architecture settings, so they may not be optimal in every scenario
- Ghidra and Ninja are excluded here, as they cannot identify variadic functions or their positions at all

Evaluation: Parameter Identification (Table 4)

Table 4: Results for Parameter Identification.

Arch	Opt	Precision				Recall			
		IDA	Ghidra	Ninja	CDA	IDA	Ghidra	Ninja	CDA
x64	O0	68.42	71.86	71.95	77.81	70.53	71.19	74.17	79.33
	O1	68.00	67.80	64.11	76.50	59.20	67.47	64.32	75.38
	O2	75.38	74.22	66.47	79.12	73.97	73.44	66.47	77.73
	O3	73.86	73.28	65.91	78.84	72.42	71.97	65.86	77.22
x86	O0	88.66	84.22	85.34	92.03	89.38	84.56	85.95	88.47
	O1	88.92	70.64	73.66	92.29	90.86	73.56	74.8	93.51
	O2	90.41	75.87	75.67	92.41	92.51	77.51	77.21	93.83
	O3	86.73	71.32	73.85	92.01	88.98	72.59	74.82	92.59
AArch64	O0	88.31	82.33	50.23	90.50	89.22	83.92	47.74	90.88
	O1	88.39	85.27	48.50	89.92	89.45	83.98	46.01	90.28
	O2	88.50	83.41	54.38	90.22	91.44	83	53.87	92.36
	O3	90.88	87.34	47.47	92.41	92.43	86	45.19	93.34
Arm	O0	87.85	86.88	83.95	89.40	89.88	88.02	85.69	90.89
	O1	90.62	88.38	85.26	91.76	91.83	86.05	86.4	91.34
	O2	84.89	86.67	79.71	89.44	85.67	84.89	79.77	89.36
	O3	87.03	87.55	80.85	91.94	87.41	85.05	80.83	91.4
Mips	O0	81.92	61.62	61.07	89.15	84.59	59.78	58.69	89.72
	O1	79.78	45.92	53.96	87.00	80.66	41.2	50.05	87.41
	O2	62.23	59.76	62.56	72.17	63.14	59.89	62.72	72.67
	O3	60.17	57.28	61.18	70.20	60.54	57.11	61.22	70.3
Avg.		81.55	75.08	67.30	**86.26**	82.20	74.56	67.09	**86.40**

Findings

- CDA achieves the highest average precision and recall: 86.26% and 86.40%
- IDA is reasonable everywhere but drops on Mips (71.02 / 72.23); Ghidra and Ninja degrade on Mips and AArch64
- CDA is notably stronger at O2 and O3, where baselines fluctuate most (x86 and Mips)

Evaluation: Return Value Detection (Table 5)

Table 5: Results for Return Value Detection.

Arch	Opt	Precision				Recall			
		IDA	Ghidra	Ninja	CDA	IDA	Ghidra	Ninja	CDA
x64	O0	62.68	70.50	75.56	75.81	48.21	75.04	42.8	75.74
	O1	63.61	60.88	74.15	77.77	59.59	66.41	46.92	77.70
	O2	62.80	64.83	75.88	79.67	52.11	69.35	48.71	79.43
	O3	63.03	64.36	75.83	79.59	52.89	69.28	48.1	79.73
x86	O0	80.31	82.41	87.44	88.44	66.53	87.09	44.95	88.18
	O1	80.44	70.93	85.19	92.55	68.41	78.78	42.38	89.95
	O2	81.18	72.77	85.85	93.34	65.99	79.61	35.10	90.85
	O3	81.73	72.52	85.77	92.94	68.06	79.58	41.20	90.06
AArch64	O0	75.19	54.33	68.17	64.39	63.48	58.50	71.97	69.72
	O1	73.53	42.66	61.89	81.26	66.15	50.90	72.08	67.49
	O2	71.70	46.01	63.62	80.16	64.49	53.06	70.56	83.08
	O3	74.08	40.35	58.36	80.55	67.11	47.72	71.00	83.85
Arm	O0	78.6	42.47	48.55	80.42	67.00	46.45	37.62	80.53
	O1	72.59	41.09	57.30	80.12	67.12	54.59	39.65	82.6
	O2	74.78	53.94	63.84	82.34	68.30	60.77	47.90	83.83
	O3	76.41	44.17	53.42	83.82	69.45	50.25	38.28	84.98
Mips	O0	78.29	83.97	87.09	82.03	60.62	81.79	58.65	85.3
	O1	76.12	87.34	86.60	88.1	65.35	86.86	68.19	89.08
	O2	53.72	57.48	70.98	70.20	45.44	71.10	49.75	72.87
	O3	51.20	69.70	70.06	69.59	44.34	69.89	47.33	71.75
Avg.		71.60	61.14	71.78	**81.15**	61.53	66.85	51.16	**81.34**

Findings

- Precision gains of 9.55 / 20.01 / 9.37 over IDA / Ghidra / Ninja, and recall gains of 19.81 / 14.49 / 30.18
- Averages: CDA 81.15 precision and 81.34 recall, against a best baseline of 71.78 and 66.85
- One exception: AArch64–O0 recall is slightly below Ninja, because Ninja is assigned a low prior (0.55) there while performing well
- No single base decompiler wins this task: Ninja has the highest precision, Ghidra the highest recall — yet CDA combines both

Evaluation: Parameter Type Recovery (Figure 9)

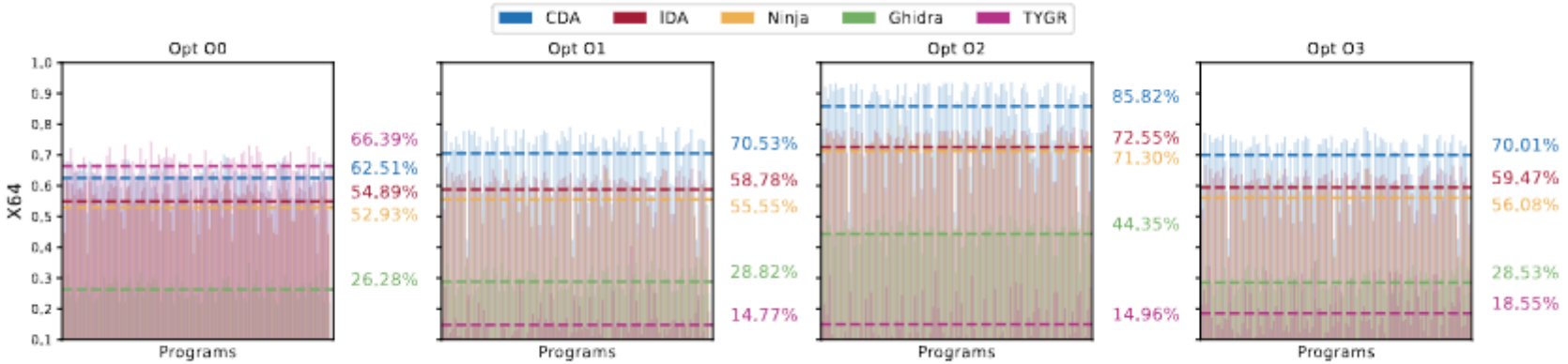


Figure 9: Type Accuracy between Different Tools.

Findings

- At O0 all tools are comparable (52.93%–66.39%), with TYGR highest at 66.39% — AI methods exploit the rich semantics of unoptimized binaries
- CDA is superior at higher levels: 70.53% at O1 and 85.82% at O2, outperforming baselines by 11.75–41.71% across O1–O3
- TYGR degrades sharply at O1–O3 because it works directly on assembly, where optimization introduces subtle variations, and because its data-flow methodology cannot handle direct parameter transfers through registers or memory
- CDA operates on decompiled output, so it is less affected by low-level assembly changes and stays stable across levels
- Average across all levels: 72.22%

Conclusion

Contributions

- First method to recover function signatures by combining the outputs of multiple decompilers, correcting their limitations rather than replacing them
- Probabilistic constraints over factor graphs at the decompilation level, plus three merge rules (direct, supersede, downgrade)
- Evaluated on 43,944 binaries and 10M+ functions across 5 architectures and 4 optimization levels, with the lowest time and memory cost among the compared tools

Personal Thoughts

- Idea that decompiler errors are structured, so a failure becomes usable evidence with no training, no labels, and no GPU is the new aspect to think of.
- There is no user-defined structure recovery and eight of the twenty-one rules are never shown firing on real code would be the limitation.
- The next step of this research could be making the rules fire in the real code, or making a model that would actually reflect this idea.

Thank You