```
mov eax, DWORD PTR [rbx]
call <unsafe_fn>
add rsp, 0x40
ret
```

RUBY

Unmasking Unsafe Rust in Stripped Binaries via Machine Learning

Xiang Cheng · Sangdon Park · HyungSeok Han · Xiaokuan Zhang · Taesoo Kim

Georgia Tech · POSTECH · Microsoft · George Mason University

DSN ’26

The first tool to locate unsafe regions in Rust binaries — no source code required

BACKGROUND

Rust Safety Model: Safe vs Unsafe

Ownership + borrowing give compile-time memory safety — but strict rules push developers into unsafe escape hatches.

Safe Rust — 76.3%

Compiler-checked: ownership, borrowing, lifetimes. Never causes undefined behavior.

Unsafe Rust — 23.7%

Developer-checked: raw pointers, FFI, inline asm, unions, mutable statics.

360+

memory-safety bugs in Rust (5 yrs)
— almost all rooted in unsafe

24.6%

of ecosystem code is direct unsafe,
often pulled in via dependencies

MOTIVATION

The unsafe Keyword Vanishes at Compilation

- Auditors of proprietary firmware and drivers only have the binary
- Source tools (Rudra, RPG, ERASAN) all need source code
- Rustc drops unsafe info early (MIR → LLVM IR)

```
1 // a is &i32, dereferencing is safe
2 *a;          // => ; deref op is optimized out
3 // a is *i32, dereferencing is unsafe
4 unsafe {*a} // => mov eax,DWORD PTR [rbx]
```

Listing 1 — safe deref vs. unsafe deref: one instruction apart

Goal: recover the **direct unsafe operations** — call to unsafe function: X , actual unsafe operation: O

MOTIVATION

Why It's Hard

1. **12 diverse unsafe operations, each a subproblem**
2. **Safe and unsafe often differ by a single instruction**
3. **Architectures and LLVM versions reshape the code**

```
<reference>:
    ... ; omitted 13 instructions for simplicity
    ; deref reference is optimized and removed
    8eb4: add    rsp,0x40
    8eb8: pop    rbx
    8eb9: ret

<pointer>:
    ... ; omitted 13 instructions for simplicity
    8f33: mov    eax,DWORD PTR [rbx]; deref ptr
    8f35: add    rsp,0x40
    8f39: pop    rbx
    8f3a: ret
```

Listing 5 — reference deref is optimized away as register operations whereas; the raw-pointer dereference survives as an explicit memory load

KEY INSIGHT

Compiler Optimizations Leave Fingerprints — Learn Them

Safe Rust gives the compiler more constraints, so safe and unsafe code compile **subtly differently** — differences a model can learn at ecosystem scale.

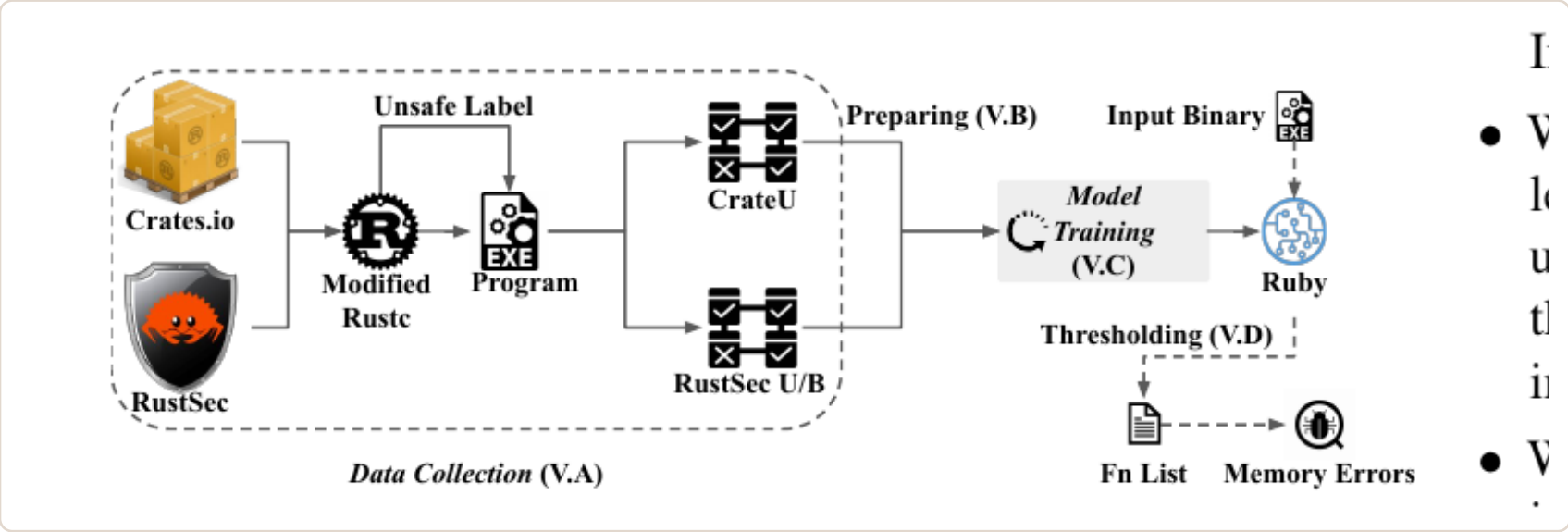

Fig. 1 — RUBY's workflow: label crates.io with a modified Rustc, train a classifier, threshold with a statistical guarantee

Output: a prioritized function list — e.g., inspect 7.43% of the program, catch ~89% of the bugs.

SETTING

Threat Model

Given

- Stripped release binary — no symbols, no source
- Function boundaries (IDA / Ghidra)

Find

- Direct unsafe operations — where undefined behavior can originate
- As a small sub-region → targets for symbolic execution & directed fuzzing

Twelve Unsafe Operations in Rustc

Label	Unsafe Operations	Description	% of functions
1	`CallToUnsafeFunction`	Call an unsafe function. This type has two subtypes:	17.61%
2		"internal" (label 1) and "external" (label 2)	0.17%
3	`UseOfInlineAssembly`	Use a `asm!` macro with low-level assembly.	0.01%
4	`InitializingTypeWith`	Initialize a layout restricted type's field with a value outside the valid range.	0.56%
5	`CastPointerToInteger`	Cast pointers to integers in const functions. (Deprecated)	0
6	`UseOfMutableStatic`	Access to a mutable static variable.	0.08%
7	`UseOfExternStatic`	Access to a mutable static variable from external.	0.01%
8	`DerefOfRawPointer`	Dereference a raw pointer.	2.59%
9	`AccessToUnionField`	Access to a union field.	1.69%
10	`MutationOfLayoutConstrainedField`	Change the layout of a constrained field. (Deprecated)	0
11	`BorrowOfLayoutConstrainedField`	Borrow a layout constrained field. (Deprecated)	0
12	`CallToFunctionWith`	Call to a function that requires special target features.	1.02%

Table I — the 12 unsafe operations Rustc checks, and their share of all functions across crates.io

- Calls to unsafe functions dominate (17.8%) — but they are symptoms; RUBY targets the direct operations that root-cause them
- Three phantom ops (5, 10, 11) never appear in 150K real crates → excluded

Skip the Call, Find the Root Cause

- Detecting 'calls an unsafe function' directly is intractable (wrappers, unbounded call context)
- Direct Unsafe Delegation: every unsafe function must ground its unsafety in a non-call direct unsafe operation

```
pub unsafe fn set_len(..)         { self.len = new_len; }    // indirect — delegates
pub unsafe fn get_unchecked(..)   { .. }                     // indirect — delegates
fn get(..) -> Option<&T> {
    unsafe { Some(slice_get_unchecked(slice, self)) }      // DIRECT — RUBY's target
}
```

Root-cause instructions are exactly where memory corruption is physically possible — ideal targets for downstream fuzzers.

Each Operation Leaves a Trace

Raw pointer deref (8)

References get promoted to registers; raw-pointer loads survive

```
mov eax, DWORD PTR [rbx]
```

Inline assembly (3)

Hand-written asm is preserved verbatim, unlike optimized Rust

```
asm!("shl {tmp},1", ...)
```

Union access (9)

Operand widths reveal which typed field is touched

```
DWORD vs QWORD PTR [rsp]
```

Mutable statics (6–7)

<GLB>/<EXT> metadata tokens expose .data/.bss and FFI use

```
<GLB> mov [static], eax
```

Constrained types (4)

Niche optimization shrinks layouts — sizes betray the type

```
Option<NonZeroI64> = 8 B
```

HW features (12)

SIMD / AES-NI intrinsics emit distinctive instructions

```
aesenclast xmm0, ...
```

APPROACH

Datasets

CrateU — training

- Modified Rustc labels unsafe locations across all 107,460 crates
- 10M functions sampled (1:1 safe/unsafe) from ~903M

RustSecU / B — evaluation

- 121 real root-cause bugs from 5 years of RustSec advisories
- 257 crates, all excluded from training — zero leakage

NEEDLE IN A HAYSTACK

1.8M

functions in RustSecB

301

of them are buggy

< 0.02%

bug rate

APPROACH

Model & Trustworthy Thresholding

1. **Embed —** static-analysis tokens: <GLB> globals, <EXT> external calls
2. **Tokenize —** custom assembly tokenizer; long functions segmented
3. **Classify —** RoBERTa-large, multi-label root-cause output
4. **Threshold —** PAC conformal bound picks $\hat{t}$ with a recall guarantee

Ask for 90% recall (ε = 0.1) …

91.75%

recall of unsafe functions delivered
(guarantee holds w.p. $1 - 10^{-3}$)

A worklist with a statistical recall contract — not a heuristic cutoff.

0.98 AUPRC — Even on Never-Seen Crates

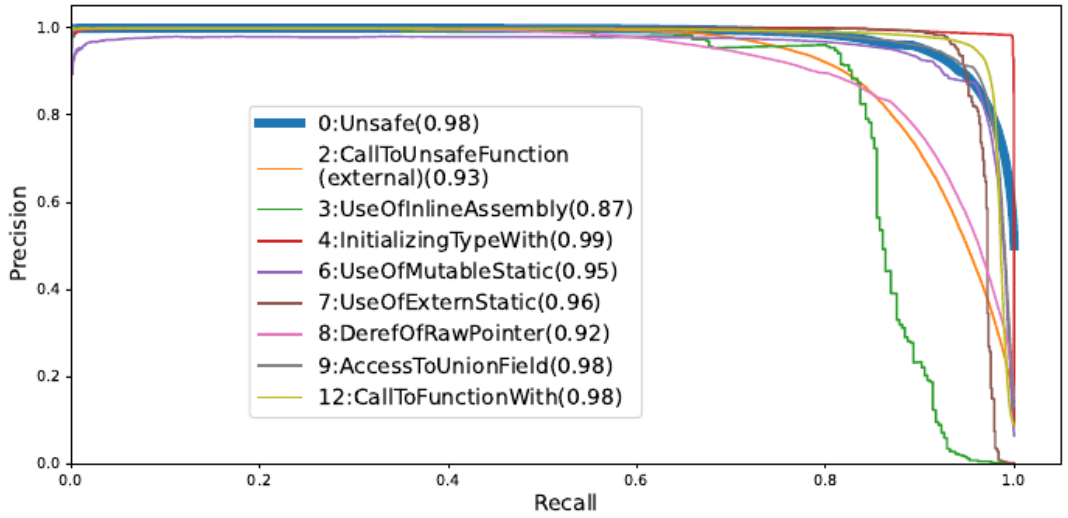

Fig. 2 — CrateU test set (x64): 0.98 overall AUPRC, high scores on every unsafe label

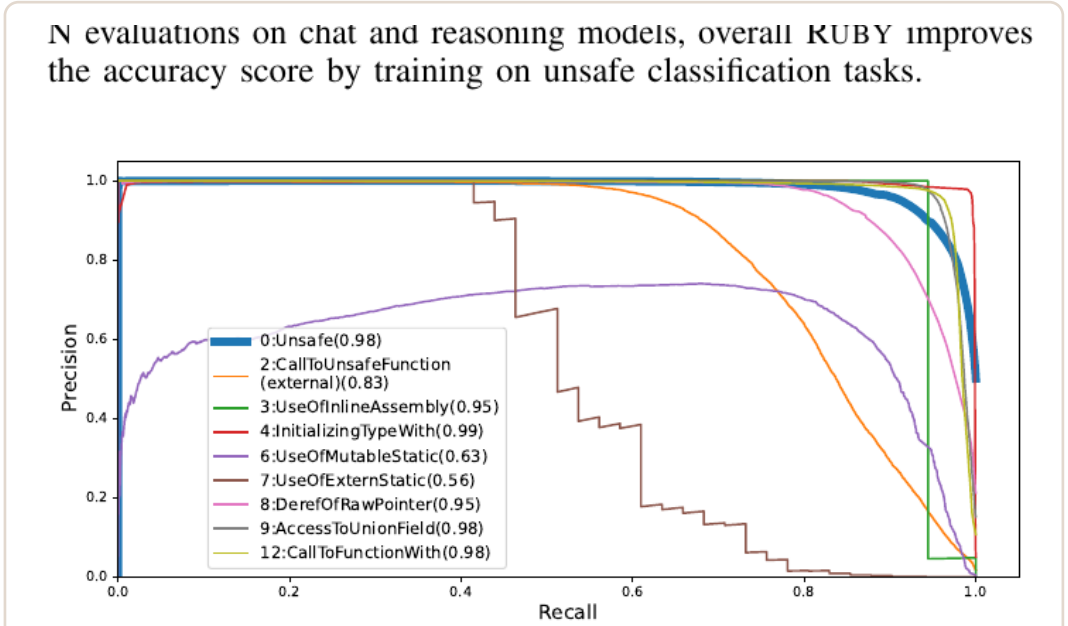

Fig. 3 — RustSecU (excluded from training): 0.98 AUPRC, precision 95.2%, recall 91.1%

Unsafe blocks are identifiable in Rust binaries — and the signal generalizes to unknown assembly.

A Dedicated Model Beats SOTA LLMs

Model	K=1			K=3			K=5		
Accuracy % / N=?	1	3	5	1	3	5	1	3	5
Sonnet-4.5	12.3	13.7	13.4	24.0	25.7	22.6	32.0	31.1	30.0
Gemini3-flash	38.6	41.1	41.7	61.4	62.3	62.0	63.7	64.3	**65.1**
GPT-5-chat	43.4	48.3	48.9	55.4	57.7	57.8	58.8	62.6	63.4
Opus-4.6	49.7	50.3	49.4	59.7	60.3	60.9	65.4	65.7	**66.1**
Geimin3-pro	33.1	47.1	52.5	49.7	56.6	62.6	47.4	55.7	60.6
GPT-5.2-pro	44.0	47.7	49.1	54.0	58.3	59.7	57.7	62.3	62.6
GPT-4.1-FT	58.0	62.3	64.0	64.0	69.1	72.3	64.3	69.2	**74.6**
RUBY	83.1								

Table III — few-shot (K) and best-of-N accuracy of chat, reasoning, and fine-tuned models vs. RUBY

83.1%

RUBY accuracy on unsafe classification

+20 pts

over the best LLM configuration (GPT-4.1 fine-tuned, 74.6%)

EVALUATION · RQ2

Shrinking the Search Space

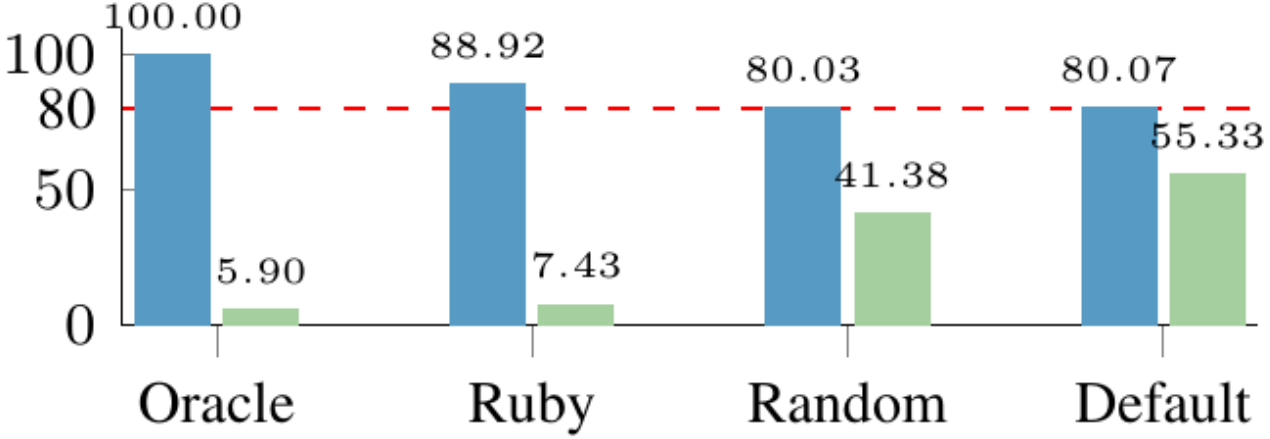

Fig. 4 — at the same 80% target recall (blue), RUBY inspects far fewer functions (green, lower is better) than random or default ordering

7.43%

of functions to inspect,
88.9% bug recall inside

1.25×

the coverage of the source-code oracle — 4–6× better than the rest

Throughput: all of crates.io (~100K binaries) analyzed in about a week.

EVALUATION · RQ2

Million-Line Real-World Binaries

deno

JS/TS runtime · 9.36M LOC

0.907

AUPRC

servo

Web browser · 11.10M LOC

0.890

AUPRC

artichoke

Ruby interpreter · 1.93M LOC

0.839

AUPRC

rustpython

Python interpreter · 8.02M LOC

0.832

AUPRC

None in the training set · each >1M LOC · analyzed in under three hours (Table IV).

Robust Across Toolchains & Architectures

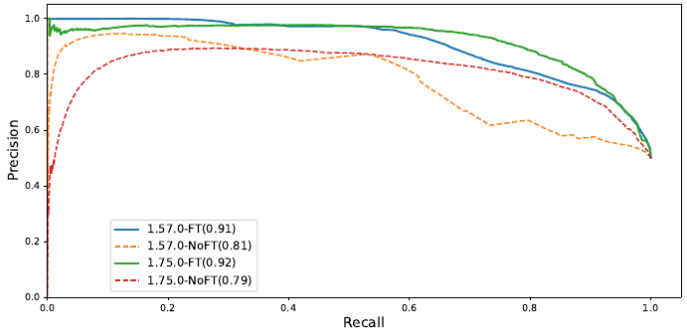

Fig. 5 — new Rustc/LLVM versions: <1 hr of fine-tuning restores 0.91–0.92 AUPRC

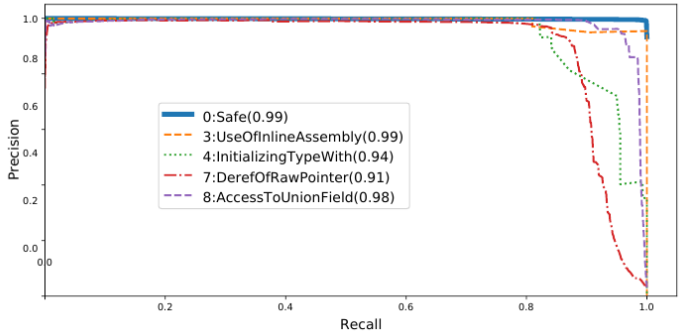

Fig. 6 — ARM matches x64: unsafe is lost before codegen, so the signal is portable

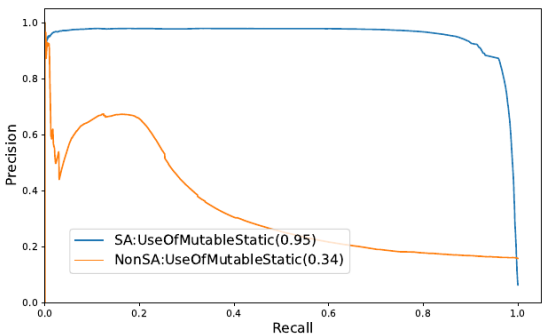

Fig. 7 — ablation: static-analysis tokens add +0.61 AUPRC

Guiding Symbolic Execution (Angr)

RUSTSEC	Name	x86_64			ARM		
		Baseline	Oracle	RUBY	Baseline	Oracle	RUBY
2021-0015	search_error	7060.73	660.14	1607.65	29028.36	3491.61	1465.16
	excel_to_csv	4668.81	343.36	424.62	2548.93	6750.60	343.76
2020-0043	bench-server	43200(TO)	2908.19	20565.21	22305.18	19151.26	6781.42
	external_shutdown	43200(TO)	18805.00	39182.24	N/A		
2021-0009	crosstalk	26816.39	244.69	5601.20	N/A		
2021-0088	worldbank	1641.82	2264.12	3269.44	3161.93	3725.85	2972.57
2021-0092	extension1	8672.45	263.82	506.95	3439.36	317.63	616.76
	extension2	10425.88	1351.28	24621.30	4495.87	1391.97	4837.85
	stream	6283.93	4.70	423.35	2385.22	887.84	484.90
2021-0094	predefined	6551.97	5050.39	4598.43	14804.18	495.10	17458.97
	file_watcher	27198.87	3982.78	11730.89	15422.56	3811.30	13806.29
2021-0090	texture	43200(TO)	8618.49	4396.34	N/A		
2021-0085	binjs_dump	11034.37	481.09	546.64	33386.22	771.72	589.74
	binjs_decode	43200(TO)	3043.78	1603.98	3368.33	2636.86	2404.86
average(seconds)		20225.37	3430.13	8505.59	12213.29	3948.34	4705.66

Table V — time (s) to find each RustSec bug: no guidance vs. source-code oracle vs. RUBY, on x64 and ARM

−57.95%

analysis time vs. baseline
(−61.4% on ARM)

1.48×

the oracle's time — without ever seeing source code

Directed Fuzzing & 5 New Android Bugs

Issue	Target	Baseline	RUBY Guided	Ratio
	rmpv	7.80	6.20	79.4%
	asn	86.78	35.04	40.4%
	tiff	158.44	132.58	83.7%
	serde	5081.94	4931.55	97.0%
	proc_macro2	10360.74	8048.14	77.7%
	boa	27791.16	26474.50	95.3%
	cpp_demangle	31128.52	30317.32	97.4%
	average	10659.34	9959.88	78.7%

Table VI — AFLGo with RUBY's targets reaches the same crashes 21.26% faster on average

Android 16 Rust libraries

- 93 crates, 1,534 API functions
- AFL++ black-box fuzzing on RUBY's top-50 targets

5 bugs

confirmed & patched by Google
· 74.6% less fuzzing time

DISCUSSION

Limitations

Inherited

- Blind to whatever Rustc can't label
- ML is lossy: FPs/FNs vs. the source oracle

Implementation

- Trained on 10M of ~903M records (hardware limits)
- RoBERTa prototype — context splits hurt long functions

CONCLUSION

Takeaways

First of its kind	Recovers unsafe regions from stripped Rust binaries by learning compiler fingerprints.
Accurate & guaranteed	0.98 AUPRC with a PAC recall guarantee — ~20 pts ahead of SOTA LLMs.
Practically useful	−57.95% symbolic execution, −21.26% fuzzing time; 5 Android bugs patched by Google.

Artifact: doi.org/10.5281/zenodo.14217066