SCRIBE: Practical Static Binary Patching via Binary-Aware Recompilation of Decompiled Code

Euro S&P 2026

Han Dai, Soumyakant Priyadarshan, Abdullah Imran, Ruoyu Wang, Antonio Bianchi

Purdue University, Arizona State University

Weekly Meeting Seminar: Jiyong

Fixing Vulnerabilities when…

- No Source?
  - Closed-source, abandoned, or vendor-unsupported COTS binaries
- No Build Environment?
  - Outdated / irreproducible toolchains, heavily customized libraries

- **Binary Patching via Decompilation**
  - Decompile the target function, patch it in C, recompile, inject it back

Types of Binary Patching

- **Direct Binary Rewriting**
  - Edit instructions in place; space handled by trampolines / new segments
  - Needs assembly-level expertise: calling conventions, registers, layout
- **Lifting and Reassembly**
  - Lift the whole binary to LLVM IR or reassemblable assembly, then rebuild
  - Scales poorly on large binaries; still a low abstraction level
- **Both: hard to apply complex semantic patches written for C code**

Decompilation-based Patching

- **Binary → Decompile → Patch in C → Recompile → Link back**
  - Work with familiar source code and reuse existing patch diffs
  - OSSPatcher does this, but assumes matching source and toolchain

- **Critical challenge: decompiler inaccuracies**
  - Syntactic — the code does not compile at all (~50% of Hex-Rays output)
  - Semantic — it compiles, but behaves differently from the original

Syntactic Issues

- **Language-rule violations — the compiler aborts before any object file**

- Malformed code
  - Bad declarations (int[5] name), illegal characters (. @ $)
  - Non-standard keywords: __noreturn, __usercall, __thiscall
- Missing definitions
  - Decompiler-invented types and macros (BYTE, DWORD, __readfsqword) have no implementation; structs, enums, and globals are undefined
- Prototype mismatches
  - Call site and callee disagree on argument count or return type (e.g. reading a value from a function declared void)

Semantic: Memory Layout Distortion

- **Code compiles and looks right, but memory is arranged differently**

- Decompilers rarely recover struct definitions
  - Members become separate local variables; object bounds on the stack are hard to infer
- The compiler is then free to reorder, re-pad, and re-align them
  - Example: a 64-byte blake2b_param struct comes back as 7 char arrays (v4–v10)
  - Recompiled, they are scattered — only the first 4 bytes are initialized, so the hash breaks
- Same problem for bool sizes, stack canaries, and spill slots
  - A single byte of offset error means corruption, not a visible error

Linking Recompiled Functions

- **The recompiled function must coexist with an untouched binary**

- No room to grow
  - A patched function is usually larger, but moving any other function or data object would break every existing code and data reference
- Calls in must keep working
  - The function may be reached through pointers (e.g. virtual calls), and pointers cannot be reliably found in a stripped binary
- Calls out must keep working
  - It has to reach original functions and globals at their exact addresses

SCRIBE: SCRIBE Can Recompile and Inject Binary Edits

- **Minimally invasive: recompile only the functions that change**

- Decompiler
  - Syntax fixes + metadata from the original binary → C1
- Recompiler
  - Binary-aware compilation: enforce the original stack layout → C2
- Patcher
  - Resolve symbols and retrofit the object file into the binary → C3

- All fixes applied to every function — no issue detection needed

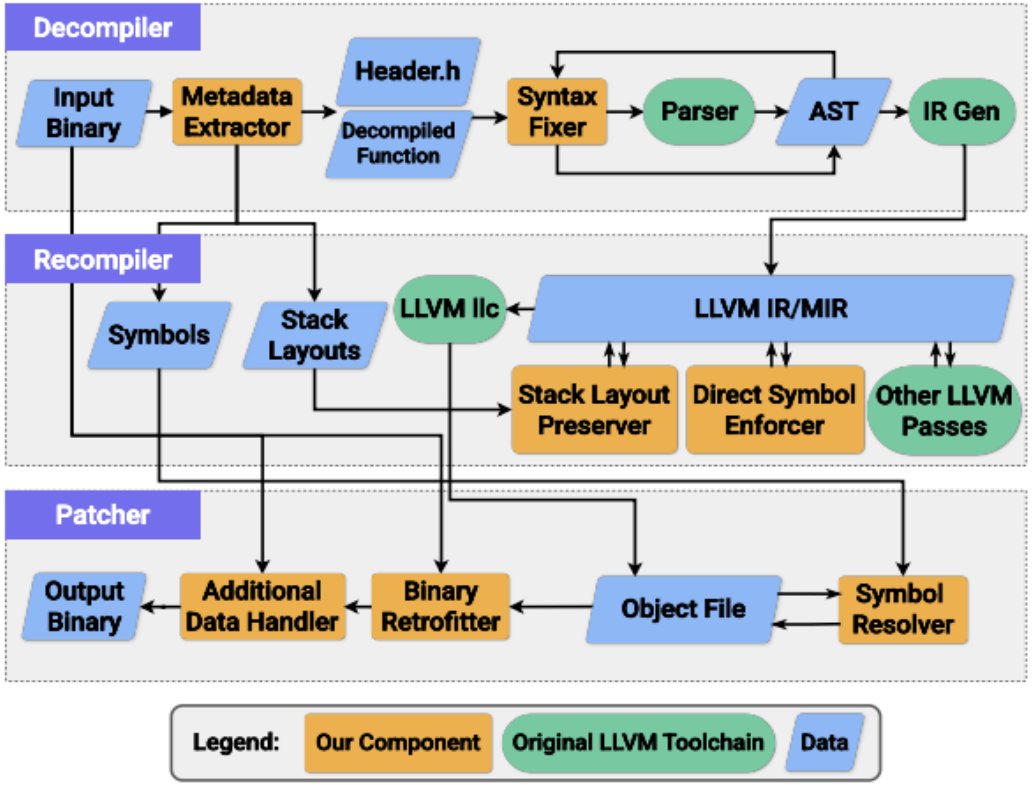

Figure 2: Overview of SCRIBE

Decompiler: Syntax Fixer

- Normalize names, drop non-standard keywords
- Supply the missing definitions through a crafted header
  - Decompiler macros become inline assembly, keeping features like stack canaries intact
- Fix prototypes by overestimating, never underestimating
  - Never mark a non-void function void; decompile callees first (Hex-Rays) or in address order (Ghidra)
  - Patched Clang AST gives each call site its own signature; strict argument-count checks relaxed
- **Metadata Extractor** — function boundaries, stack-variable offsets, addresses of globals and external functions, global type definitions

Recompiler: Stack Layout Preserver

- **Goal: the recompiled function must use the same stack layout as the original**

- The problem: a compiler decides where local variables live
  - It picks sizes, order, and padding on its own — so recompiled code puts variables at different offsets than the original binary expects
  - Normally that is fine; here it silently corrupts memory
- The fix: change the compiler, not the code
  - Added compiler passes intercept the stage that assigns stack offsets, before it computes them

Patcher: Symbol Resolver

- **Problem: the rest of the binary has no object files to link against**
  - Auto-generated linker script maps each symbol to its binary address (printf = 0x401050)
  - External library calls still routed through the original GOT
- **Binary Retrofitter — layout preserving**
  - In-place if it fits → reuse padding → else add a new segment
  - A trampoline jmp at the old address keeps direct calls and function pointers valid
- **Additional Data Handler**
  - New jump tables, string literals, and globals injected at the right offsets

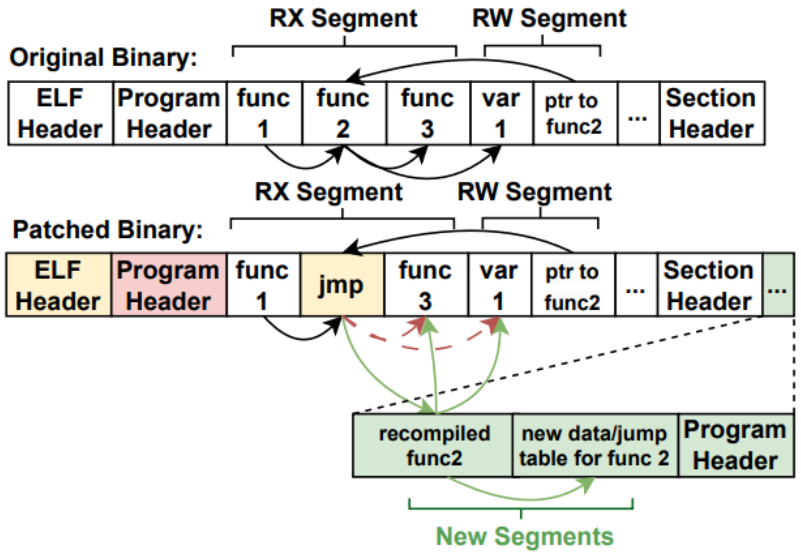

Figure 3: Overview of our binary rewriting technique for preserving symbolic references

Recompilability: Syntactic Fixes

- **Setup**
  - Coreutils 9.4 + DARPA CGC, built at -O0 to -O3
  - Hex-Rays (IDA 8.4) and Ghidra 11.2
  - Every function decompiled, recompiled, re-injected, then run against the test suite
- **Recompilable after syntax fixes**
  - Hex-Rays: 84.1% of functions
  - Ghidra: 49.4% — loose C conformance (e.g. 7-bit int7) creates a long tail of one-off issues

Decompiler	Dataset	Opt Lvl.	Total Functions (TF)	Recompiled Funcs. (Recomp)
Hex-Rays	Coreutils	-O0	18,775	16,977 (90.4%)
		-O1	17,249	14,933 (86.6%)
		-O2	14,504	10,618 (73.2%)
		-O3	14,575	10,026 (68.8%)
	CGC	-O0	4416	4119 (93.3%)
		-O1	3882	3581 (92.2%)
		-O2	3730	3277 (87.9%)
		-O3	3801	3059 (80.5%)
Hex-Rays Average			N/A	84.1%
Ghidra	Coreutils	-O0	19,314	13,086 (67.8%)
		-O1	20,289	10,816 (53.3%)
		-O2	19,735	8984 (45.5%)
		-O3	19,909	8947 (44.9%)
	CGC	-O0	4686	2747 (58.6%)
		-O1	4613	2213 (48.0%)
		-O2	4693	1846 (39.3%)
		-O3	4760	1793 (37.7%)
Ghidra Average			N/A	49.4%

Recompilation (Hex-Rays)

- **Per-binary view: how much of each binary can be recompiled**

- Cases above 80% success: 80 → 394 (of 640)
- Cases below 60%: 235 → 7
- 12 CGC binaries fully recompiled at -O0; 56 more above 90%

- **Beyond targeted patching, this hints at whole-binary hardening and instrumentation**

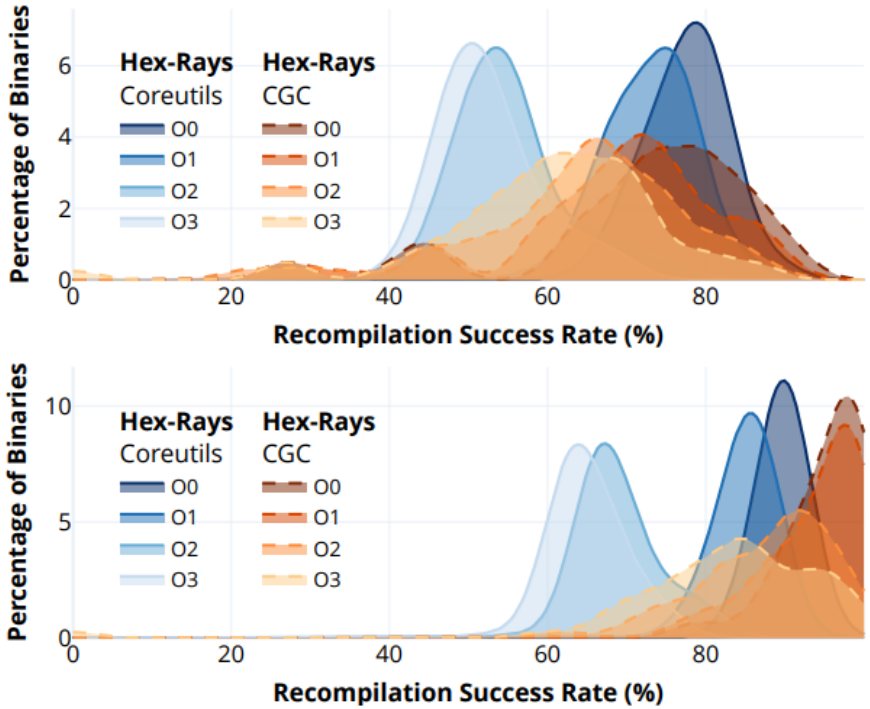

Figure 4: Hex-Rays Recompilation Success Rate before (top) and after (bottom) applying SCRIBE's fixes

Passing Tests: Semantic Fixes

- **80.9% of failing Hex-Rays functions fixed**

TABLE 1: Recompilation results with Ghidra and Hex-Rays. SCRIBE achieves successful recompilation for 84.1% of Hex-Rays and 49.4% of Ghidra functions (Recomp). Semantic errors occur in 17.9% (Hex-Rays) and 6.1% (Ghidra) of functions (SemError), of which SCRIBE fixes 80.9% and 53.3% respectively (FixedSem over SemError).

Decompiler	Dataset	Opt Lvl.	Total Functions (TF)	Recompiled Funcs. (Recomp)	Passing Tests w/o Fixes (TestPass)	Failing Tests w/o Fixes (SemError)	SCRIBE Fixed Funcs. (FixedSem)	SCRIBE Fixed % (FixedSem over SemError)
Hex-Rays	Coreutils	-O0	18,775	16,977 (90.4%)	14,532 (77.4%)	2445 (13.0%)	2209 (11.8%)	90.3%
		-O1	17,249	14,933 (86.6%)	12,553 (72.8%)	2380 (13.8%)	1934 (11.2%)	81.3%
		-O2	14,504	10,618 (73.2%)	8294 (57.2%)	2324 (16.0%)	1750 (12.1%)	75.3%
		-O3	14,575	10,026 (68.8%)	7868 (54.0%)	2158 (14.8%)	1635 (11.2%)	75.8%
	CGC	-O0	4416	4119 (93.3%)	3256 (73.7%)	863 (19.5%)	755 (17.1%)	87.5%
		-O1	3882	3581 (92.2%)	2724 (70.2%)	857 (22.1%)	755 (19.4%)	88.1%
		-O2	3730	3277 (87.9%)	2401 (64.4%)	876 (23.5%)	666 (17.9%)	76.0%
		-O3	3801	3059 (80.5%)	2284 (60.1%)	775 (20.4%)	566 (14.9%)	73.0%
Hex-Rays Average			N/A	84.1%	66.2%	17.9%	14.5%	80.9%
Ghidra	Coreutils	-O0	19,314	13,086 (67.8%)	11,466 (59.4%)	1620 (8.4%)	777 (4.0%)	48.0%
		-O1	20,289	10,816 (53.3%)	9911 (48.8%)	905 (4.5%)	548 (2.7%)	60.6%
		-O2	19,735	8984 (45.5%)	7984 (40.5%)	1000 (5.1%)	526 (2.7%)	52.6%
		-O3	19,909	8947 (44.9%)	7887 (39.6%)	1060 (5.3%)	540 (2.7%)	50.9%
	CGC	-O0	4686	2747 (58.6%)	2388 (51.0%)	359 (7.7%)	192 (4.1%)	53.5%
		-O1	4613	2213 (48.0%)	1893 (41.0%)	320 (6.9%)	168 (3.6%)	52.5%
		-O2	4693	1846 (39.3%)	1590 (33.9%)	256 (5.5%)	138 (2.9%)	53.9%
		-O3	4760	1793 (37.7%)	1539 (32.3%)	254 (5.3%)	138 (2.9%)	54.3%
Ghidra Average			N/A	49.4%	43.3%	6.1%	3.2%	53.3%

Real World Applicability

- 14 reproducible CVEs (2 coreutils, 12 binutils)
  - 15 functions recompiled; one CVE needed two distinct functions
  - Function size from <100 to >2,000 LoC (avg. 379, median 277)

- **13 / 14 successfully patched — all test cases pass, PoC no longer works**
  - The one failure is a function-pointer corner case, not yet implemented

TABLE 3: Real-world CVEs evaluated with SCRIBE. LOC indicates lines of code in the patched function. CWE(s) indicates Common Weakness Enumeration classification(s). Res. indicates result (Y=success, N=failure).

CVE-ID	Dataset	Program	CWE(s)	LOC	Res.
CVE-2014-9471	coreutils	date	N/A	336	Y
CVE-2024-0684	coreutils	split	787, 122	139	Y
CVE-2020-16590	binutils	readelf	415	274	Y
CVE-2020-16591	binutils	readelf	125	274	Y
CVE-2020-16592	binutils	nm	416	118	Y
CVE-2020-16593	binutils	addr2line	476	370	Y
CVE-2020-16599	binutils	nm	476	64	Y
CVE-2021-20284	binutils	nm	787, 119	243	N
CVE-2021-20294	binutils	readelf	787	140	Y
CVE-2022-35205	binutils	readelf	617	836	Y
CVE-2022-35206	binutils	readelf	476	2145	Y
CVE-2022-4285	binutils	nm	476	282	Y
CVE-2022-48063	binutils	objdump	400	112	Y
CVE-2023-1972	binutils	objdump	787, 119	286	Y

Usability

- **Human study: 18 participants, 3.5 h, 3 CVEs (easy / medium / hard)**
  - Each vulnerability patched twice — with and without SCRIBE, order randomized, 30 min cap
  - With SCRIBE: 18/18 on all three, ~7 min average · Without: 0/18, 2/18, 0/18 ($p < 0.00001$)

- **LLM agents: GPT-5, Claude 4.5 Sonnet, Gemini 2.5 Pro**
  - 27/27 trials with SCRIBE in 12–40 s; 0/27 without it
  - Humans and LLMs failed the same way: a correct patch abandoned after unexplained test failures

TABLE 2: Comparison of participants' success in patching vulnerabilities with and without SCRIBE

Vulnerability	W/O SCRIBE	With SCRIBE
Easy Vulnerability	0/18	18/18
Medium Vulnerability	2/18	18/18
Hard Vulnerability	0/18	18/18

CVE-2020-16593

- **binutils null-pointer dereference**
  - Source fix is one extra null check
- The var / spec_var structs are never recovered
  - Member access shows up as raw pointer arithmetic on stack variables
  - Naive recompilation shifts the offsets, so the patched binary reads the wrong memory
- **SCRIBE recompiles the 370-line function with layout preserved**
  - All 18 participants patched it, several with no decompiler experience

```
  if (var->name == NULL)
    var->name = spec_var->name;
- if (var->file == NULL)
+ if (var->file == NULL && spec_var->file != NULL)
    var->file = strdup (spec_var->file);
```

```
  if ( !*((_QWORD *)v27 + 4) )
    *((_QWORD *)v27 + 4) = v37[4];
- if ( !*((_QWORD *)v27 + 2))
+ if ( !*((_QWORD *)v27 + 2) && v37[2] )
  {
    v15 = strdup((const char *)v37[2]);
    *((_QWORD *)v27 + 2) = v15;
```

Listing 4: Source (top) and decompiled (bottom) patches for CVE-2020-16593

Remarks

- Paper introduces an intriguing perspective into where recompilable decompilation can be useful
- It is slightly unclear where and how they extract metadata
  - For instance, function boundary is taken from the binary
  - Is the binary stripped? Or do we utilize debug symbols?
- Stated limitations worth noting
  - Correctness is only output equivalence on existing test suites
  - Evaluated on x86-64 only; patches spanning multiple functions untested