B2R2 in Action: From Vulnerability Analysis to Binary Repair

Sehoon Kim, Jung Hyun Kim, Seung Il Jung, Jun Tae Kim, HyungJoon Yoon, and Sang Kil Cha
KAIST
Daejeon, South Korea
{slatfatf, jhkim, sijung, gnrkr789, topcue1019, sangkilc}@softsec.kaist.ac.kr

***Abstract*—Diagnosing vulnerabilities and assessing repairs are essential software maintenance tasks. When source code or a build environment is unavailable, engineers must work directly with binaries, combining code inspection, execution, and patching while preserving analysis context. We demonstrate B2R2, an open-source binary-analysis framework that supports this iterative maintenance workflow. First presented in 2019 with a focus on efficient binary parsing, disassembly, and lifting to an intermediate representation, B2R2 now also supports control-flow graph recovery, symbolic execution, concrete execution through emulation, and binary patching. Transformer, a new interactive tool in B2R2, organizes analysis operations into small actions that users compose through a domain-specific language inspired by Unix pipelines. Dynamic typing supports generic actions across data types, simplifying workflow composition over B2R2's strongly typed API. Type-guided tab completion helps analysts build and revise these workflows in a console. We demonstrate this workflow on CVE-2023-43641 in `libcue`, using symbolic execution to generate a triggering input, disassembly to identify the root cause, and an assembly-level patch to repair the binary. We then rerun the same input on the patched binary in B2R2's emulator to check whether the failure persists. The tool is available at https://github.com/B2R2-org/B2R2, and the demo video is available at https://youtu.be/S27GC0hw0VI.**

***Index Terms*—binary analysis, reverse engineering, domain-specific languages, context-aware completion, software security**

I. Introduction

Maintaining deployed software can require diagnosing and repairing vulnerabilities in third-party or legacy components whose source code or build environment is unavailable. Binary analysis enables software engineers to investigate these components and assess changes directly in the deployed binaries.

Developing a binary patch requires understanding the vulnerability and assessing how the proposed change affects program behavior. Analysts may need to inspect disassembly, trace control flow, test a failure condition, and assess the effects of a change, connecting evidence across these operations and representations. Results inform subsequent steps, and new findings can prompt a return to earlier decisions [20]. An effective binary-analysis environment should make this iterative process easy to explore, inspect, and revise.

Existing binary-analysis frameworks offer many of the required capabilities, but assembling them into a coherent workflow remains a practical challenge. When operations do not share analysis results and context, analysts must connect them through scripts or manually transfer the information needed by subsequent steps. For example, an analyst might investigate a suspected vulnerability in `IDA Pro` [8], examine its failure condition with symbolic execution in `angr`, and return to IDA to patch and inspect the binary. Connecting these steps requires preserving relevant addresses, execution assumptions, and findings throughout the workflow.

We demonstrate how B2R2 [9] supports this iterative workflow, from vulnerability diagnosis to binary repair. First presented in 2019, B2R2 initially focused on an efficient frontend for binary parsing, disassembly, and lifting to an intermediate representation. It has since evolved to support control-flow graph recovery, code inspection, symbolic execution, among other capabilities, with diverse applications in published research (see §II). Transformer, B2R2's interactive tool, brings these capabilities into a single console application through a pipeline-based domain-specific language. B2R2 and Transformer are intended for software engineers and security analysts who investigate vulnerabilities and assess binary repairs, as well as researchers who develop and evaluate binary-analysis workflows.

Transformer's central design idea is to build complex binary-analysis workflows from small, composable operations, which we call *actions*. Inspired by the Unix approach of connecting small utilities through pipes, Transformer provides a domain-specific language in which pipe operators pass the result of one action to the next. Analysts can thus break a task into manageable steps and express the connections between them directly in a pipeline. Intermediate results can also be named and retained, allowing analysts to inspect a result, reuse it in another pipeline, or revise a later step.

B2R2's strongly typed API requires analysts to understand and work with analysis-specific types and interfaces when connecting operations, which can make workflow construction demanding. Transformer lowers this barrier through a dynamically typed DSL that supports generic actions across different data types and representations. This flexibility lets analysts reuse common operations across diverse analysis results and compose workflows with less type-specific integration code.

Alongside this flexibility, Transformer uses action type information to guide workflow construction in its interactive shell. As an analyst builds a pipeline, the shell uses the available result type and partial command syntax to suggest compatible actions and guide parameter entry through tab completion. The interface therefore helps analysts discover

available operations and supply their parameters as they explore a workflow. Transformer's key innovation is the integration of generic action composition through dynamic typing, reusable intermediate results, and type-guided completion into a single interactive language for binary analysis. This integration lets users build workflows incrementally: they compose a step, inspect its result, and extend or revise the workflow while specifying analysis choices such as target regions and execution conditions.

We demonstrate Transformer on CVE-2023-43641 [7], using symbolic execution to generate a triggering input and disassembly inspection to identify the root cause. We then apply an assembly-level patch and rerun the same input in B2R2's emulator to check whether the failure persists, connecting diagnosis, patching, and validation in a single interactive workflow.

This paper makes the following contributions:

- We present B2R2's evolution from its 2019 frontend-focused release into a framework supporting control-flow graph recovery, code inspection, symbolic execution, and rewriting.
- We introduce Transformer, an interactive tool that simplifies B2R2's strongly typed API through dynamically typed action pipelines, reusable intermediate results, and type-guided completion.
- We demonstrate a workflow for CVE-2023-43641 that combines symbolic input generation, disassembly inspection, assembly-level patching, and concrete execution to assess the patch against the triggering input, all within one console.

II. B2R2 and Transformer

B2R2 was first presented in 2019 with a focus on an efficient binary-analysis frontend that parses binary files, disassembles machine code, and lifts instructions into an intermediate representation [9]. Since then, B2R2 has expanded to support control-flow graph recovery, symbolic execution, concrete execution through emulation, and binary patching. It has also gained a graphical user interface for interactive binary analysis.

B2R2 has also served as a foundation for research, with at least eight papers using or building on the framework for vulnerability discovery [6], [5], binary rewriting [10], function identification [11], [14], decompilation [13], exploit specification [12], and binary dataset construction [21]. Prior studies have demonstrated B2R2's performance advantages in binary parsing and lifting to an intermediate representation [9], as well as its accuracy in recovering function boundaries and control-flow graphs from stripped binaries [14], [11], [10]. At the time of writing, B2R2's GitHub repository has over 550 stars and 60 forks [1].

This paper focuses on Transformer, a new component of B2R2 that lets users compose its analysis capabilities through a domain-specific language (DSL) for binary analysis within a single interactive console. The DSL combines *actions* that perform individual operations, *pipelines* that connect them, and *bindings* that retain results for reuse. Figure 1 provides an overview of how these constructs organize the underlying API calls into an interactive analysis workflow.

TABLE I
Representative built-in Transformer actions.

Actions	Role
`@load`, `@slice`, `@grep`, `@list`	Load, select, search, and enumerate binary information.
`@disasm`, `@lift`, `@recover-cfg`, `@asm`	Disassemble and lift code, recover CFGs, and assemble instructions.
`@run-concrete`, `@run-symbolic`	Run concrete and symbolic execution.
`@edit`, `@compare`, `@print`	Modify and compare binaries, and display results.

Existing frameworks expose analysis capabilities through mechanisms such as Ghidra's scripting APIs [16], `angr`'s Python APIs [19], and BAP's analysis passes and plugins [3]. Transformer's contribution is to combine direct composition and reuse of analysis values with type-guided completion in a dynamically typed DSL. This integration lets users build workflows incrementally without connecting framework API calls themselves.

A. Actions and Pipelines

An action is the basic unit of computation in the DSL: it encapsulates an analysis operation with typed input and output interfaces. Action names are prefixed with `@`, and parameters specify choices such as the binary to load or the region to inspect. Table I summarizes several representative actions and their roles. These actions cover code inspection and search, concrete and symbolic execution, and binary modification and comparison.

The pipe operator `|>` passes the result of one action to the next, forming a pipeline that makes the data flow between operations explicit. Two actions can be connected when the output type of the first is compatible with the input type of the second. For example, the pipeline `@load path=<binary> |> @slice section=.text |> @disasm` first loads a binary, selects its code section, and then disassembles the selected code. The pipeline thus expresses the analysis steps and their dependencies without requiring users to manage the underlying API calls.

B. Bindings and Interactive Reuse

Users evaluate these pipelines in Transformer's read–eval–print loop (REPL). To carry a result from one command to another, the DSL provides bindings, which associate names with evaluated results and retain them for the session. Subsequent commands can refer to a bound name to inspect the result or use it as input to another pipeline without rerunning the actions that produced it.

The `target = @load path=<binary>` command in Figure 3 illustrates this mechanism: `target` retains the loaded binary for use in both disassembly and symbolic execution. Figure 4 reuses `target` in an editing pipeline

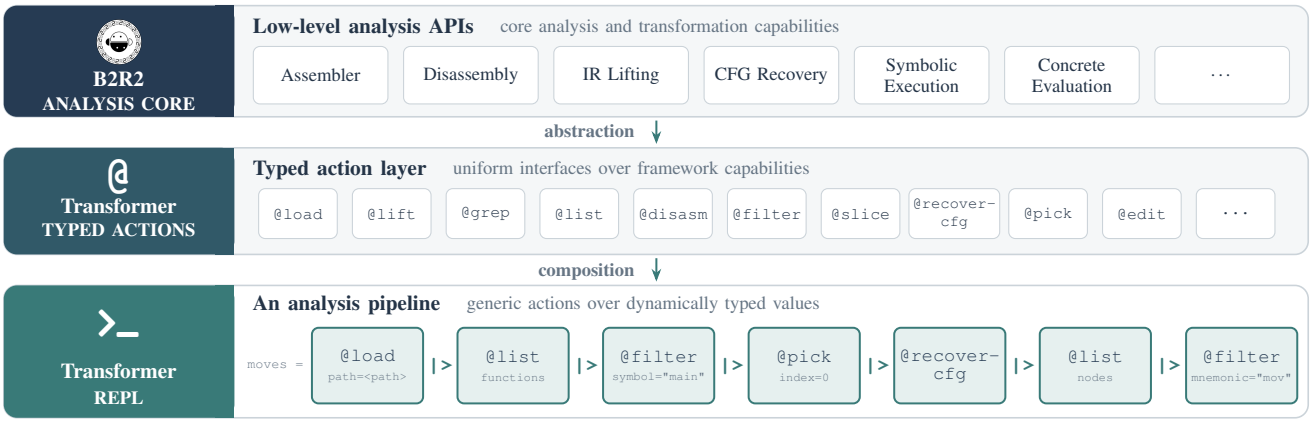

Fig. 1. Transformer abstracts B2R2's low-level analysis APIs into uniform action interfaces. The pipeline loads an unstripped binary, picks the first function with the `main` symbol, recovers its CFG, and selects nodes containing `mov` instructions. The generic `@filter` action first filters a function collection by `symbol` and later filters CFG nodes by `mnemonic`. The resulting `moves` binding can be reused in subsequent analyses. Transformer's context-aware completion guides users toward compatible action compositions.

and binds the result to `patched`. The pipeline `(target, patched) |> @compare` then compares the patched binary with the original. Together, pipelines and bindings let users build an analysis across successive REPL commands, inspect intermediate results, and explore alternative next steps while reusing earlier work.

C. Context-Aware Completion

To help users compose these workflows, Transformer's REPL provides context-aware tab completion. As users enter a command, the shell uses the available result type and partial command syntax to suggest compatible actions and guide parameter entry. Users can therefore discover applicable operations as they build a pipeline, without memorizing all action names or their parameter syntax. This guidance lets users learn the DSL incrementally while composing binary analysis scripts, choosing steps according to their goals.

III. Demonstration: From Diagnosis to Repair

We combine Transformer's actions, pipelines, and bindings in a vulnerability diagnosis and repair workflow. We generate a bug-triggering input, identify the root cause, and patch the binary, then replay the input in concrete execution to check whether the failure persists. For this workflow, we use CVE-2023-43641 in `libcue` [15]. As Figure 2 shows, `track_set_index` checks whether `i` exceeds `MAXINDEX` but permits negative values, allowing an indexed write before the start of the array.

A. Repair Scenario

Consider an analyst tasked with repairing a crash in a deployed third-party library when its source code and build environment are unavailable. A crash report, together with symbol or debugging information, identifies the affected function and its address range, but the analyst still needs to find a triggering input, diagnose the root cause, and assess a binary patch. Our demonstration follows this workflow,

```
1  void track_set_index(Track *track, int i, long ind)
2  {
3    if (i > MAXINDEX) {
4      fprintf(stderr, "too many indexes\n");
5      return;
6    }
7    track->index[i] = ind;
8  }
```

(a) Source code.

```
1  0x401e1f:  cmp dword ptr [rbp - 0xc], 0x63
2  0x401e23:  jle 0x401e4a
```

(b) Bounds check.

```
1  0x401e4a:  mov rax, qword ptr [rbp - 0x8]
2  0x401e4e:  mov edx, dword ptr [rbp - 0xc]
3  0x401e51:  movsxd rdx, edx
4  0x401e54:  lea rcx, [rdx + 0x10]
5  0x401e58:  mov rdx, qword ptr [rbp - 0x18]
6  0x401e5c:  mov qword ptr [rax + rcx * 8 + 0x8], rdx
```

(c) Indexed write.

Fig. 2. Vulnerable `track_set_index` code and corresponding instructions. (a) Source code. (b) The bounds check for `if (i > MAXINDEX)`. (c) The indexed write for `track->index[i] = ind`.

treating `libcue` as a binary-only target despite its open-source availability. Starting from the identified function, we use under-constrained symbolic execution [17] to find an input that reaches a memory-policy violation, then investigate and repair the faulty code. The source in Figure 2 is provided only to explain the diagnosis to readers; Transformer operates on the binary.

B. Demonstration

Figure 3 loads the binary and disassembles the affected function (lines 1–3), then marks `ESI` and `RDX` as symbolic and searches for a memory-policy violation (lines 4–8). The resulting model assigns `ESI = 0x80000001` (-2,147,483,647 as a signed 32-bit integer) and `RDX = 0x0`. The disassembly reveals why this input triggers the fault: the signed `jle` branch admits the negative index, allowing an out-of-bounds write (Figure 2(b)–(c)).

```
target = @load path=<binary>
target |> @slice start=0x401e08 end=0x401e63
|> @disasm |> @print
target
|> @run-symbolic
  entry=0x401e08
  sym-regs=[ ESI; RDX ]
  ptr-regs=[ RDI=0x400:rw ] |> @print
```

Fig. 3. A Transformer script excerpt for vulnerability diagnosis.

```
patched =
  target
  |> @edit replace
    start=0x401e23 asm="jbe +0x27"
(target, patched) |> @compare |> @print
patched
|> @run-concrete
  entry=0x401e08
  regs=[ RSI=0x80000001; RDX=0x0 ]
  ptr-regs=[ RDI=0x400:rw ] |> @print
```

Fig. 4. A Transformer script excerpt for repair and validation.

Figure 4 replaces the conditional branch with `jbe +0x27` and displays the binary difference (lines 9–13). Replaying the triggering input on the patched binary reaches the function's return without a memory-policy violation (lines 14–18). A symbolic re-query, not shown here, could establish the absence of such violations within the execution model if it proves the query unsatisfiable. A video demonstrating this workflow is available at https://youtu.be/S27GC0hw0VI.

C. Planned Validation Study

The case study demonstrates an end-to-end workflow, but does not measure the effort required for users to construct and revise such workflows. We plan to conduct a user study to evaluate how Transformer's typed pipelines and context-aware completion support these tasks.

IV. Related Work

Existing binary-analysis frameworks support workflow construction through project models, scripting APIs, and interactive commands. `Ghidra` [16] retains analysis results in a project model accessible through its GUI and scripting APIs, while `angr` [18] exposes binaries, execution states, and analysis results as Python objects for symbolic execution and exploration. `radare2` [2] provides a command language with sequencing, filtering, and pipes that send command output to external programs; r2pipe bindings also expose command results, including JSON, to scripts. `BAP` [4] combines project-level analysis passes, a Knowledge Base, and Primus for execution analysis; its Knowledge Base REPL provides completion and contextual hints for analysis queries [3].

Transformer's pipelines pass analysis values directly between actions, and bindings retain these values for reuse within the REPL. Its dynamically typed DSL supports generic actions across data types, while completion uses result types to suggest applicable next steps. The distinction lies in combining direct composition and reuse of analysis values with type-guided completion in the DSL, simplifying workflows over B2R2's strongly typed API.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-01332, Developing Next-Generation Binary Decompiler and No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains) and the KAIST Quantum+X Convergence R&D Project.

References

[1] "B2R2," https://github.com/B2R2-org/B2R2.

[2] "Radare2," https://github.com/radare/radare2.

[3] Binary Analysis Platform, "BAP documentation," https://binaryanalysisplatform.github.io/bap/api/master/, 2026.

[4] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of the International Conference on Computer Aided Verification*, 2011, pp. 463–469.

[5] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *Proceedings of the International Conference on Automated Software Engineering*, 2021, pp. 227–239.

[6] J. Choi, K. Kim, D. Lee, and S. K. Cha, "NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2021, pp. 1973–1989.

[7] CVE Program, "CVE-2023-43641," https://www.cve.org/CVERecord?id=CVE-2023-43641, 2023.

[8] Hex-Rays SA., "IDA Pro," https://www.hex-rays.com/products/ida/.

[9] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha, "B2R2: Building an efficient front-end for binary analysis," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

[10] H. Kim, S. Kim, and S. K. Cha, "Towards sound reassembly of modern x86-64 binaries," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025, pp. 1317–1333.

[11] H. Kim, J. Lee, S. Kim, S. Jung, and S. K. Cha, "How'd security benefit reverse engineers? the implication of intel CET on function identification," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2022, pp. 559–566.

[12] J. H. Kim, S. Gustaman, and S. K. Cha, "PoE: A domain-specific language for exploitation," in *Proceedings of the Silicon Valley Cybersecurity Conference*, 2024, pp. 1–6.

[13] J. H. Kim, S. Kim, J. Choi, and S. K. Cha, "EVMpress: Precise type inference for next-generation evm decompilation," in *Computer Security. ESORICS International Workshops*, 2025, pp. 344–360.

[14] S. Kim, H. Kim, and S. K. Cha, "FunProbe: Probing functions from binary code through probabilistic analysis," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2023, pp. 1419–1430.

[15] libcue Contributors, "libcue: CUE sheet parser library," https://github.com/lipnitsk/libcue.

[16] National Security Agency, "Ghidra software reverse engineering framework," https://github.com/NationalSecurityAgency/ghidra, 2019.

[17] D. A. Ramos and D. Engler, "Under-Constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium*, 2015, pp. 49–64.

[18] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "(state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2016, pp. 138–157.

[19] UCSB SecLab, "Angr," https://github.com/angr/angr.

[20] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' processes," in *Proceedings of the USENIX Security Symposium*, 2020, pp. 1875–1892.

[21] H. J. Yoon and S. K. Cha, "Binforge: Facilitating generation of diversified binary datasets," in *Proceedings of the Asia-Pacific Software Engineering Conference*, 2026, p. to appear.