Deterministic Decompilation of Hermes Bytecode for React Native Application Analysis

Anthony Bondu · *Symbiotic Security*

Abstract. Applications built with React Native ship their JavaScript compiled to Hermes bytecode [1]. Inside the app package this logic lives in one bundle file, `index.android.bundle` on Android and `main.jsbundle` on iOS. It holds tens of thousands of functions in a register based instruction set. The compiler discarded the local variable names, and the Metro bundler [2] flattened the file boundaries. A reviewer who opens such a bundle sees raw bytes, or at best a faithful disassembly that is still far from source. We describe a decompiler that turns Hermes bytecode back into readable JavaScript. It recovers structured control flow, rebuilds Metro modules as ECMAScript modules with import and export statements, and assigns identifiers from two sources: function names read straight from the binary, and local names inferred from usage. The system is rule based and deterministic, so a bundle always yields the same output and every step can be audited. It covers sixty bytecode versions, from HBC 40 to HBC 99, which span React Native releases from 2019 to 2026. Correctness is checked by a public round trip corpus that recompiles and re executes decompiled programs across eleven compiler versions (359 programs, all passing), and the tool has been used in real penetration testing and in capture the flag challenges. The implementation is about 46,000 lines of Rust and is open source at https://github.com/SymbioticSec/hermes-decomp.

1 Introduction

React Native lets a single JavaScript code base drive native mobile applications, and since 2019 its default engine has been Hermes, a bytecode interpreter written by Meta for fast startup on constrained devices [1]. When a release build is produced, the application JavaScript is compiled ahead of time into Hermes bytecode (HBC), a compact instruction format the engine runs in place of the source text, and packaged inside the app. The entire program is one asset, `index.android.bundle` in the Android package and `main.jsbundle` in the iOS one, which a production build compiles with optimizations and without debug information. The bytecode format is the same on both platforms, so the analysis that follows applies to either.

For anyone auditing such an application the practical consequence is severe. The source that implemented authentication, key handling, request signing or license checks is no longer present. What remains is a register machine program: a table of functions, each a stream of instructions over virtual registers, which are the numbered storage slots the interpreter uses in place of named variables, a shared string table, and metadata describing the module layout produced by the Metro bundler, the build tool that merges a React Native project's many source files into this one file. Local variable names are gone because the compiler assigns registers, not names. Module boundaries are gone because Metro rewrites every file into a call to a runtime registration function and concatenates the results.

The tools available today stop at disassembly. Projects such as hbctool [3], hermes-dec [4] and hasmer [5] parse the container and print an instruction listing, which is accurate but keeps the analyst at the level of individual opcodes. Reading control flow, following a value across basic blocks, or telling which module a call reaches all remain manual. The gap between a disassembly and the JavaScript a reviewer wants to reason about is exactly the work a decompiler performs.

This paper presents such a decompiler for Hermes bytecode. Our goals are readability of the output, breadth across bytecode versions, and determinism. By readable we mean output a developer can follow: loops and conditionals written as such rather than as numbered jumps, functions and variables that carry names rather than register numbers, and each original source file recovered as its own module, in place of one flat list of instructions. The last of these goals separates the work from the current research direction in decompilation, which is dominated by large language models that translate assembly to source [6] [7]. Those systems produce fluent output but are probabilistic: the same input can yield different results, and a plausible looking line may not match the binary. For security review this is a poor trade. We instead recover structure by program analysis, so that every control construct and every function name in the output has a traceable origin in the input, and the tool can be pointed at a bundle repeatedly with byte identical results.

The contributions are the following. We give the full pipeline that takes a Hermes bundle to JavaScript, including the parsing of sixty on disk bytecode layouts and the reconstruction of the modern function header, twelve bytes wide, introduced at HBC 97. We describe the control flow recovery, with attention to a loop shape that Hermes emits through loop inversion, a transform that moves the exit test to the bottom of the loop, which naive structuring turns into invalid code. We describe the reconstruction of Metro modules into

ECMAScript modules. We separate identifier recovery into a deterministic layer, where names are read from the binary, and a heuristic layer, where local names are guessed, and we argue this separation is the honest way to report naming. Finally we validate behaviour with a round trip corpus across eleven compiler versions, and we report output quality measured on a real application bundle of 32,322 functions. Beyond these controlled measurements, the tool has been used to analyze real applications during penetration tests and capture the flag events, a use we describe in a public write up [8].

2 Background

2.1 The Hermes bytecode container

A Hermes bundle begins with a header carrying a magic number and a version word. The version is the single most important field for a decompiler: it selects the opcode table and the layout of every following structure. Our implementation ships sixty opcode tables, one per version from 40 to 99, generated from the engine sources, and dispatches on the version at parse time.

After the header come the function headers and the string table. Every string in the program is stored once here, and a parallel kind table records what each entry is used for: a plain string literal, an identifier such as a variable or property name, or a predefined name known to the engine. The distinction matters to us because identifiers are the strings that can become names in the output. Then come an array buffer and an object key and value buffer, which hold the constant literals the code builds objects and arrays from, tables of the compiled regular expressions and of the BigInt literals, and, in builds that keep it, debug information. Each function header records the entry offset, the parameter count, the frame size, the environment size and the number of exception handlers, together with a set of flags. Two header shapes exist. The legacy small header packs these fields into a fixed layout and stores an `info_offset` that points at overflow data including the exception handler table. From HBC 97 onward the header is twelve bytes and the `info_offset` is gone; the exception table instead follows the large header overflow, aligned to four bytes. Handling both shapes is necessary to reconstruct `try` and `catch` on recent React Native releases.

2.2 Register based instructions

Hermes instructions operate on virtual registers rather than a stack. A function declares a frame of some size and the instructions read and write register slots within it. There is no notion of a named local at this level. A source assignment `let x = a + b` becomes an arithmetic opcode writing a register, and the name `x` exists only in debug information, which release builds omit. This is why recovering local names is an inference problem and not a lookup, a point we return to in Section 7.

2.3 Metro modules

Metro, the React Native bundler, does not preserve files. Each source module is wrapped in a call to a registration function, conventionally written `__d(undefined, factory, moduleId, dependencyMap)`, where the factory is a function of the fixed parameters `(global, require, module, exports)`. All modules are concatenated into one bundle and a runtime require resolves numeric module identifiers. A decompiler that ignores this structure produces one flat sea of functions; recovering it restores file boundaries, import edges and the export surface of each module.

3 System overview

The decompiler is a pipeline of passes over a typed intermediate representation, an internal tree of statements and expressions that sits between the raw bytecode and the final JavaScript and that every pass reads and rewrites. Figure 1 lists the stages. Parsing yields the container structures. A per function stage lifts instructions to a control flow graph and recovers structured statements. A sequence of transforms simplifies expressions, reconstructs language level patterns, rebuilds modules and assigns names. A code generator emits JavaScript.

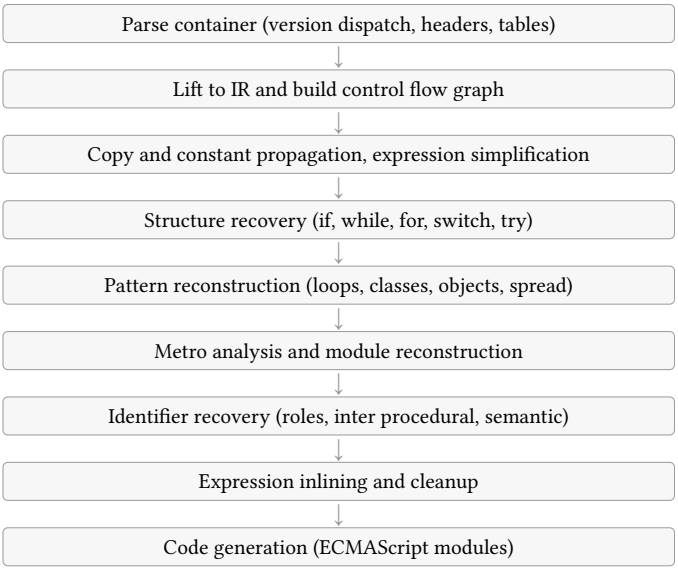

Figure 1: The decompilation pipeline. Each stage consumes and produces the same intermediate representation until code generation. Simplified: live range splitting sits inside propagation, Metro detection runs a second time on the raw program, and naming is itself several passes (register roles, inter procedural, semantic), covered in Sections 5 to 7.

One design decision shapes the rest of the system. Module detection runs on the program before optimization, because copy propagation and inlining rewrite the `__d` call pattern that the Metro detector matches on. The pipeline therefore lifts the bytecode twice: once in a raw form used only to recognize

modules, and once in the optimized form used for the final output. The two passes are kept separate on purpose.

4 Control flow recovery

4.1 From instructions to a graph

Lifting walks a function once, translating each instruction into an IR statement or expression and recording jump targets. Conditional and unconditional jumps partition the instruction stream into basic blocks, which are straight line runs of instructions with no branch in the middle, and the jumps between blocks give the control flow graph, the graph whose nodes are those blocks and whose edges are the possible jumps. Exception handlers contribute additional edges: the entry of a catch block is a control flow root, because it is reachable only through the exception mechanism and not by a normal edge, and treating it as a root is what lets later analyses compute dominators and a postorder that include the handler.

4.2 Structuring

Structuring turns the graph back into nested statements. We compute the immediate post dominator of each two way branch to find the point where its arms rejoin, and emit an `if` whose two arms are the code dominated by each successor up to that join. Loops are found from back edges: an edge whose target dominates its source, meaning every path that reaches the source first passes through the target, marks a natural loop whose header is the target. A loop with a single exit becomes a `while` or, once an update statement is identified at its tail, a `for`. Switches are recovered from the dense jump tables Hermes emits for them. This follows the line of work on structuring compiled code without goto statements [9] [10], adapted to the specifics of Hermes.

4.3 A loop shape that breaks naive structuring

Hermes compiles counted loops with loop inversion: the loop test is emitted twice, once as a guard before the body and once at the tail as the back edge condition. For a source loop over an array the bytecode has the shape `i = 0; if (i < n) { do { body; i++ } while (i < n) }`. A structuring pass that finds the join point of the header branch without accounting for the back edge places the merge one block too early, folds the loop tail into the wrong arm, and emits a `break` that sits outside any loop together with a counter that is never initialized. The result parses in some cases but does not run. Figure 2 shows the graph: the test `i < n` appears twice, once at the guard and once at the latch that closes the back edge, and it is this duplicated test that a naive pass mistakes for two separate branches.

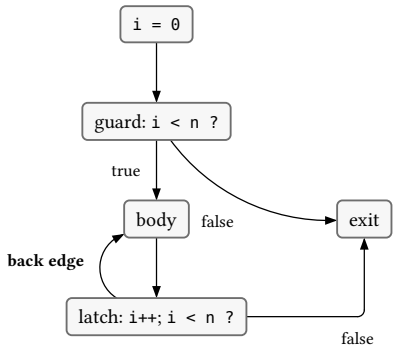

Figure 2: Control flow of an inverted counted loop. The loop test is present at both the guard and the latch, the latter closing the back edge. A structuring pass that does not account for the back edge places the join too early and folds the tail into the wrong arm.

We correct this in three steps. First, the reachability walk that computes the merge point stops before re entering the branch block, so the back edge no longer pulls the join earlier. Second, the copy that carries the loop counter across the latch, which Hermes emits as a move in one block and an increment in another, is resolved by a cross block copy propagation guarded by reaching definitions, so the increment reads the counter and not an undefined temporary. Third, a loop whose header test stays inside the loop is wrapped as a bottom tested loop rather than a bare conditional. Listing 1 shows a representative function before and after these corrections.

Before: invalid, `break` outside any loop, `tmp` undefined

```
function _defineProperties(target, props) {
  let i = 0;
  if (0 < props.length) {
    let d = tmp.enumerable;   // tmp never set
    ...
    i = tmp2 + 1;             // tmp2 never set
    if (i >= props.length) break;  // no loop
  }
}
```

After: a valid counted loop

```
function _defineProperties(target, props) {
  for (let i = 0; i < props.length; i = i + 1) {
    let d = props[i];
    d.enumerable = d.enumerable || false;
    d.configurable = true;
    if ("value" in d) d.writable = true;
    Object.defineProperty(target, d.key, d);
  }
}
```

Listing 1: A counted loop from a real bundle, before and after loop recovery. Hermes emits the loop through inversion; without the correction the tail folds into the guard and the output does not run.

Once a bottom tested loop is recovered we also rewrite it toward its source form. A `while (true)` whose only exit is a trailing `if (cond) break` becomes a `do ... while`, and the guarded `do ... while` that loop inversion produces is folded back to a `for` when the guard is provably the loop test evaluated at the initial value and the counter is not read after the loop. Both rewrites are only applied when they can be shown to preserve behaviour.

5 Data flow simplification

Register machines reuse slots aggressively, so a single register can hold unrelated values at different points. Before naming, we split the live ranges of such registers, the spans over which one value is in use, using reaching definitions, the analysis that tells which assignment supplies the value read at a given point, and a union find over definitions that share a slot. The effect is a partial static single assignment form, in which each assignment produces its own name so that a reused slot no longer hides several unrelated values. On top of this, copy and constant propagation remove the temporaries that the compiler introduces for intermediate results, and an expression simplifier folds the algebra that lowering leaves behind, for example turning a negated comparison under a logical not back into the opposite comparison. These passes are what make the difference between output littered with `r3` and `tmp7` and output that reads like the loop body in Listing 1.

6 Module reconstruction

The Metro detector scans the raw lifted program for the four argument registration call and records, for each module, its numeric identifier, the function that implements it, its dependency list, and the export surface. Exports are recognized in three forms: direct assignment to a property of `exports`, a bulk assignment of an object literal to `module.exports`, and the accessor form `Object.defineProperty(exports, name, { get })` that transpilers, tools such as Babel that rewrite newer JavaScript into a form older engines accept, emit for ECMAScript named exports.

With the module graph in hand the code generator emits each module as an ECMAScript module, the module format defined by the JavaScript standard, which declares its dependencies with import and export statements. A require of a numeric identifier becomes an `import` from the resolved module name; the interop wrapper, the small helper that transpilers place around a default import to bridge the older CommonJS module style and ECMAScript modules, is recognized and collapsed to a plain default import; and the export forms above become `export` statements. The factory wrapper and its fixed parameters are dropped, so the reader sees a file with imports at the top, its functions in the middle, and its exports at the bottom, rather than a call to `__d`. Listing 2 shows a module of the real bundle before and after this step.

Metro registration (input):

```
__d(undefined, function (global, require, module,
exports) {
  var _typeof = require(4);
  var _assert = require(21);
  exports.default = function
_possibleConstructorReturn(o, r) { ... };
}, 19, [4, 21]);
```

Reconstructed ECMAScript module (output):

```
import { default as _typeof } from "_typeof";
import _assertThisInitialized from
"_assertThisInitialized";

export default function
_possibleConstructorReturn(o, r) {
  if (r) {
    return r;
  }
  if (undefined !== r) {
    throw new TypeError("Derived constructors may
only return object or undefined");
  }
  return _assertThisInitialized(o);
};
```

Listing 2: Module 19 of a real bundle. The factory wrapper and its four fixed parameters are removed, each `require` of a numeric id becomes a named `import` with the interop wrapper collapsed, and `exports.default` becomes `export default`. The function name is read from the binary.

7 Identifier recovery

Naming is where a decompiler is most tempted to overclaim, so we are explicit about what is recovered and what is guessed.

7.1 Names that are in the binary

Function names are present. A closure creation instruction, the opcode that builds a function value at run time, refers to its function by index, and that function's header carries a name field pointing into the string table. When a register is assigned such a value, we read the name and use it verbatim. This recovers the real names of transpiler helpers such as `_typeof` and `_interopRequireDefault`, and of every other named function, rather than a generic label. The name comes straight from the binary, so this layer is exact: the same bundle always yields the same function names, and each can be found byte for byte in the input.

7.2 Names that are inferred

Local variable names are not in a stripped bundle, because the compiler turned them into registers. Any name we give a local is therefore a guess. We infer these names from how each value is used, in layered passes. A register that receives an array literal, or that is the target of a `.push` call, is named `arr`. A value read from a property and then passed to a call takes

the name of that property, so an argument that came from `user.email` is named `email`. Parameters get their names from an inter procedural analysis, one that reasons across function boundaries rather than inside a single function: it collects hints at the call sites of a function and votes on the most frequent one. Callback conventions supply the rest, so that the first parameter of a `.map` callback is `item` and the first of a `.then` is `result`. These heuristics improve readability, but unlike the function names they are not guaranteed to match the source; we keep them separate from binary names throughout this paper, and Table 1 records which names are exact and which are inferred.

source	example	exact
function header	`_interopRequireDefault`	yes
debug scope (debug build)	`userEmail`	yes
call site inference	`email` from `login(email)`	no
register role	`arr` from `.push`	no

Table 1: Where a recovered identifier comes from, and whether it is guaranteed to match the source. The first two are read from the binary; the last two are inferred from usage.

The distinction is not cosmetic. On a stripped production bundle there is no source of truth for local names other than inference, and the only way to obtain exact local names is to analyze a build that kept debug information, which the tool uses when it is present. Conflating the two layers would let a decompiler present a guessed `email` with the same confidence as a function name read from the binary.

8 Evaluation

We ask two questions. Does the decompiled program behave like the original, and how much of the output is readable rather than generic. We answer the first with a round trip corpus and the second with markers counted on a real application bundle.

8.1 Round trip across compiler versions

Hermes fixes the bytecode version in the compiler binary; there is no flag to emit an arbitrary version. We therefore collect eleven Hermes compilers spanning HBC 59 to HBC 98 and compile the same suite of small self contained programs with each. A program covers one language feature and ends by printing a deterministic result: arithmetic and comparison, control flow including loops and switch and try, closures and generators, destructuring, spread and rest, classes, template literals, optional chaining. For every program and every compiler we compile to bytecode, decompile, run both the original and the decompiled output under Node.js, and compare their standard output. The harness recompiles the whole corpus with the current build on each run, so a regression in any pass shows up at once rather than hiding behind a stale artifact.

HBC version	programs	identical
59, 60, 62	93	93
71, 74, 76	99	99
83, 84, 89	99	99
96	33	33
98	35	35
total (11 versions)	359	359

Table 2: Round trip result. Every program, across all eleven compiler versions, re executes to output identical to the original.

All 359 programs re execute to identical output. The check is behavioural, not textual: it does not require the decompiler to reproduce the source, only a program with the same observable result. That is the property a reviewer actually relies on.

8.2 Output quality on a real bundle

For readability we decompile the release bundle of a shipped React Native application, a stripped production build with no debug information, holding 32,322 functions. We count markers in the emitted JavaScript. Table 3 reports the change from a baseline without the naming and loop work to the current build. Generic function labels drop by seventy seven percent as function names are read from the binary. The pass that folds a guarded do while back into a counted loop recovers several hundred `for` loops that the baseline left as flat jumps. Functions that previously emitted a `break` outside any loop fall to zero; we detect these by parsing each emitted function on its own, where such a `break` is a syntax error, so the count is a static property of the output and does not depend on running it.

marker	baseline	current
generic function label `fn`	6394	1426
recovered `for (let ...)` loops	0	438
functions with an illegal `break`	90	0

Table 3: Output markers on a 32,322 function bundle, before and after the naming and loop recovery work.

8.3 Determinism

Running the decompiler twice on the same bundle produces byte identical output. The analysis keys its on disk cache on the bytecode content, iteration order is fixed by sorted maps throughout, and no step draws on randomness or a network service. A reviewer can rerun a result and get the same file, and can diff two builds of a tool change and attribute every difference to the change.

8.4 In practice

Beyond the corpus, the decompiler has been used in penetration testing engagements and in capture the flag challenges. The target there is a real shipped bundle, and the task is concrete: find a hardcoded secret, check how certificate pinning is implemented, or read the logic behind a feature

flag. These uses exercise the tool on code the corpus does not contain, including heavily minified and obfuscated Metro output, and the recovered modules and function names are what let a reviewer reach the relevant code by hand. We offer this as evidence that the output is usable on real applications, not as a measured benchmark; the round trip corpus above is the part that is public and reproducible.

9 Related work

Structuring compiled code into readable source has a long history, from early reverse compilation [11] to methods that avoid goto statements while preserving semantics [9] [10], and to production decompilers such as the Hex-Rays decompiler in IDA [12] and Ghidra [13] for native binaries. Our control flow recovery follows this line, specialized to the loop shapes and exception layout of Hermes.

The current research emphasis is neural: systems that fine tune or prompt large language models to translate assembly or low level pseudocode into source [6] [7], some augmenting the model with control flow graphs or explicit data mappings, others adding a structure recovery phase before a naming phase. These reach fluent output and can name and type in ways a rule based system cannot, but they are probabilistic and can emit code that does not match the input. Our aim is the opposite end of the trade: deterministic output whose every element is traceable, which suits security review where a wrong but plausible line is worse than a generic one.

For Hermes specifically the existing tools are disassemblers built for patching more than for reading. hbctool [3] disassembles a supported bundle to a textual listing and reassembles an edited one, which suits changing a constant or a comparison in place, but it tracks only a few bytecode versions and stays at the instruction level. hasmer [5] offers a disassembler and assembler, with a decompiler still in progress. hermes-dec [4] produces a disassembly and a pseudo JavaScript listing that follows the opcodes rather than reconstructing source constructs. None of them recover control flow into conditional and loop statements, rebuild Metro modules, or resolve identifiers, and none report a behavioural check. Our work adds all four, across sixty versions, and validates by round trip.

10 Limitations and future work

Local names are inferred and can be wrong; we keep them separate from binary names, but a reviewer should still treat them as guesses. Some captured closures resolve to a generic label when the analysis cannot trace them to a named function, and improving that resolution is the largest remaining source of generic identifiers. The generator recovers modern generators and async functions from their compiled state machine conservatively, and complex cases fall back to a flatter form. Finally, the round trip corpus exercises language features in small programs; extending behavioural validation to whole application bundles, where no reference output exists, is open.

11 Conclusion

Hermes bytecode is the form in which most React Native applications actually ship, and until now the analyst opening one had a disassembly at best. We have shown that the bytecode can be decompiled to readable JavaScript by program analysis: structured control flow recovered from the graph, Metro modules rebuilt as ECMAScript modules, function names read from the binary and local names inferred and kept separate from them. The system covers sixty bytecode versions, and a round trip corpus across eleven compilers re executes identically for all 359 programs. We hope the separation it draws, between what is read from the binary and what is guessed, is useful beyond this engine.

References

[1] Meta. Hermes: the React Native JavaScript engine. https://hermesengine.dev, 2019.

[2] Meta. Metro: the React Native bundler. https://metrobundler.dev, 2015.

[3] P. Bong. hbctool: Hermes bytecode disassembler. https://github.com/bongtrop/hbctool, 2020.

[4] P1 Security. hermes-dec: Hermes bytecode decompiler. https://github.com/P1sec/hermes-dec, 2022.

[5] L. Baizer. hasmer: Hermes bytecode assembler. https://github.com/lucasbaizer2/hasmer, 2022.

[6] N. Jiang et al. LLM4Decompile: Decompiling binary code with large language models. 2024.

[7] CodeInverter: Control flow and data mapping augmented binary decompilation with large language models. arXiv:2503.07215, 2025.

[8] Symbiotic Security. We built a decompiler for React Native apps. https://www.symbioticsec.ai/blog/we-built-a-decompiler-for-react-native-apps-heres-how-we-tested-it-at-insomnihack, 2026.

[9] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, M. Smith. No more gotos: Decompilation using pattern independent control flow structuring and semantics preserving transformations. NDSS, 2015.

[10] K. Yakdan, S. Dechand, E. Gerhards-Padilla, M. Smith. Helping Johnny to analyze malware. IEEE S&P, 2016.

[11] C. Cifuentes. Reverse compilation techniques. PhD thesis, Queensland University of Technology, 1994.

[12] Hex-Rays. The Hex-Rays decompiler. https://hex-rays.com, 2007.

[13] National Security Agency. Ghidra: reverse engineering framework. https://ghidra-sre.org, 2019.