Lifting Hermes Bytecode to P-Code: A Ghidra Extension for Reverse Engineering React Native Apps

Martin Renze
Fraunhofer SIT
Darmstadt, Germany
ATHENE - National Research Center for Applied Cybersecurity
Darmstadt, Germany
martin.renze@sit.fraunhofer.de

Florian Magin
Fraunhofer SIT
Darmstadt, Germany
ATHENE - National Research Center for Applied Cybersecurity
Darmstadt, Germany
florian.magin@sit.fraunhofer.de

Abstract

React Native applications increasingly ship JavaScript logic as Hermes bytecode, a custom virtual-machine format that is not supported by mainstream reverse-engineering tools. We present HermesLoader, a Ghidra extension that loads Hermes bytecode, lifts it to Ghidra's P-code intermediate representation, and exposes the resulting program through Ghidra's standard disassembly, decompilation, navigation, and cross-reference workflows. The extension combines a SLEIGH specification for Hermes bytecode version 96 with a loader that reconstructs bytecode containers as typed program objects and a small set of Hermes-specific enrichment passes for strings, field-access chains, aggregate literals, and selected function references. We evaluate the implementation on source-generated Hermes bytecode samples compiled under multiple configurations. The lifter implements all 206 opcodes in version 96; our test corpus generates 199 of them and shows broad agreement with the reference `hermesc` disassembly. Remaining differences are mostly representational, with incomplete recovery of some table-driven control-flow targets as the main semantic limitation. These results show that Hermes bytecode can be integrated into Ghidra as a practical basis for interactive React Native reverse engineering.

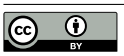
CCS Concepts

• **Security and privacy** → **Software reverse engineering**; • **Software and its engineering** → *Automated static analysis.*

Keywords

React Native, Ghidra, Lifter, Decompilation, iOS, Hermes, JavaScript, HermesLoader, P-code, SLEIGH, Reverse Engineering

ACM Reference Format:
Martin Renze and Florian Magin. 2026. Lifting Hermes Bytecode to P-Code: A Ghidra Extension for Reverse Engineering React Native Apps. In *2nd Workshop on Software Understanding and Reverse Engineering (SURE '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3847158.3847381



This work is licensed under a Creative Commons Attribution 4.0 International License.
SURE '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3019-1/2026/11
https://doi.org/10.1145/3847158.3847381

1 Introduction

Reverse engineering is a key technique for understanding the behavior, security, and provenance of modern software [2]. On mobile platforms, however, most established reverse engineering tools are designed primarily for native machine code and provide only limited support, if any, for virtual-machine-based languages and their bytecode formats. This limitation is particularly relevant for React Native applications, which frequently rely on the Hermes JavaScript engine [8, 11]. In such applications, the core application logic together with bundled JavaScript libraries is compiled into a single Hermes bytecode file [9]. This format is not directly supported by mainstream reverse engineering frameworks such as Ghidra [13], IDA, and Binary Ninja.

Several open-source tools support command-line disassembly, assembly, or pseudo-code decompilation of Hermes bytecode [1, 14, 15, 17]. However, none integrates Hermes into a general-purpose reverse-engineering framework with interactive navigation, cross-references, and IR-based analysis. Analysts therefore still lack a unified environment for interactive exploration of Hermes-based applications.

Academic Hermes analysis has also focused primarily on Android, although the bytecode is platform-independent and appears in both Android and iOS applications. We therefore target Ghidra, whose multi-platform analysis environment provides a basis for examining Hermes bytecode alongside platform-native code in future cross-language work.

We develop a Hermes bytecode lifter for Ghidra that supports interactive reverse engineering. Our approach combines a SLEIGH language specification for Hermes bytecode with a companion loader that reconstructs the bytecode container into typed memory objects, thereby enabling disassembly, P-code generation, and decompilation within Ghidra. Building on this lifter, we implement a small set of Hermes-specific enrichment passes that improve code comprehension and position the result as a foundation for downstream cross-language analysis. We evaluate the lifter for opcode coverage and disassembly agreement against the reference Hermes implementation, confirm failure-free loading on 198 Hermes-based apps from the iOS App Store, and discuss the practical utility.

Contributions

We provide the source code of HermesLoader [7]. Our artifact is publicly available at https://figshare.com/s/e7b107825b299065be91 and includes evaluation scripts/data.

- **Hermes support for Ghidra.** We implement a loader and SLEIGH lifter that reconstruct Hermes bytecode containers, enable P-code generation, and cover all 206 opcodes.
- **Analyst-oriented decompiler enrichments.** We add Hermes-specific analyses that expose strings, property access chains, aggregate literals, and selected function references to make navigation and decompiler output easier to interpret.
- **Generated evaluation corpus.** We compile targeted JavaScript samples to measure opcode coverage and disassembly agreement against the reference Hermes implementation.

2 Background

Architecture of React Native. React Native applications execute JavaScript logic in an embedded engine while rendering through native platform components. During the build, Metro combines the application code and its dependencies into one bundle. With Hermes enabled, this bundle contains compiled bytecode rather than minified JavaScript source [8, 11]. For reverse engineering, the JavaScript-side behavior is therefore concentrated in the bundle, while platform integration remains in native code.

Hermes Bytecode. Hermes is a JavaScript engine optimized for mobile and particularly for React Native [8, 11]. Hermes performs ahead-of-time (AOT) compilation of JavaScript into a compact bytecode format. This reduces startup time and memory usage.

Hermes bytecode is stored in a dedicated container format, commonly using `.jsbundle` or `.hbc` files, with sections for metadata, strings, and function bodies [9]. Each function body consists of a stream of bytecode instructions. Hermes follows a register-based virtual-machine design, in which operands typically refer to registers, constants, or entries in the string table [10].

At runtime, the application initializes the Hermes engine and loads the precompiled bytecode, which Hermes interprets to execute the application logic [10]. For analysis and reverse engineering, this format differs significantly from native ARM64 machine code because it is higher level, more regular, and closely tied to the Hermes runtime and React Native execution model.

Reversing and Lifting with Ghidra. Ghidra is an open-source software reverse engineering framework that supports disassembly, decompilation, and program analysis for a wide range of machine instruction sets [13]. Internally, Ghidra translates instructions into an intermediate representation called P-code, on which many of its analyses operate. The SLEIGH documentation describes how instruction encodings and semantics are specified and translated into P-code to enable retargetable analysis [12]. This translation from a concrete instruction set to a common intermediate representation is commonly referred to as lifting. In this setting, a loader is responsible for parsing an input file and reconstructing its contents in Ghidra's program model, whereas a lifter defines the instruction set semantics and maps individual instructions to P-code.

Since Hermes bytecode is a custom virtual instruction set, it is not supported by Ghidra out of the box. Analyzing Hermes bytecode therefore requires both components: a loader that recognizes Hermes bytecode files and maps their sections and metadata into Ghidra, and a lifter that defines Hermes instructions, their operands, and their translation into P-code.

Once Hermes bytecode has been lifted to P-code, Ghidra can apply its existing analysis infrastructure: it can reconstruct control-flow graphs, compute cross-references, and decompile Hermes functions into a higher-level representation. This makes it possible to use Ghidra's analysis workflow for Hermes-based React Native applications in a manner similar to native binaries.

3 Related Work

Decompilation of Hermes Bytecode. Several open-source tools already support textual Hermes reverse-engineering workflows, but their documented capabilities and integration targets differ. `hbctool` provides command-line disassembly and reassembly, but its documented support is limited to older Hermes bytecode versions and it does not aim to provide decompilation or interactive analysis [17]. `hermes-dec` provides a file parser, disassembler, and pseudo-code decompiler for Hermes bytecode, including support for recent HBC revisions; its documentation describes the decompiler output as pseudo-code rather than valid JavaScript source [14]. `hermes_rs` provides a maintained Rust disassembler and assembler with explicit support for multiple HBC versions, including version 96, and utilities for dumping strings, arrays, objects, and Metro module metadata; its README lists decompilation as future work [15]. `hasmer` supports disassembly to a dedicated assembly syntax and reassembly, and advertises a decompiler, but its documentation marks assembly and decompilation as work in progress and its public repository is archived [1].

Our work is closest in spirit to Falvo's Hermes decompilation plugin for Binary Ninja [3], which also aims to make Hermes bytecode more accessible for reverse engineering. A practical difference is availability and integration context: HermesLoader is built on Ghidra, an open-source analysis framework, and is released as an artifact to support reproducibility and follow-up research. Falvo's Binary Ninja plugin is not publicly available, so a direct technical comparison is not possible.

Taken together, these efforts show that Hermes disassembly, and in some cases pseudo-code decompilation, are not new as standalone workflows. Our contribution is instead to integrate Hermes bytecode into Ghidra as a first-class analysis target. By lifting Hermes instructions to P-code and reconstructing Hermes metadata as program objects, we obtain interactive disassembly, decompilation, cross-references, navigation, and a foundation for further IR-based and cross-language analysis within a framework that also supports native binaries. Our enrichment passes then improve the analyst-facing output on top of that integrated representation.

Program Analysis for React Native on Android. A core challenge in analyzing React Native apps is that app behavior is distributed across JavaScript and native components connected via framework-specific communication mechanisms. REUNIFY [4] improves analysis completeness by translating and integrating JavaScript and native-side code into an intermediate representation suitable for Soot, enabling construction of a more comprehensive behavioral model and yielding increases in call-graph reachability and taint-analysis findings on large app sets. Follow-up work [5] deepens this direction by focusing on Hermes bytecode translation to Soot's IR (Jimple), empirically characterizing opcode usage (including comparisons between benign apps and malware), introducing a

control-flow benchmark to validate lifting correctness, and explicitly modeling cross-language communication to further expand native-side coverage. Collectively, these efforts demonstrate that React Native–aware lifting and cross-language modeling are essential for achieving meaningful static-analysis coverage on Android.

While Hermes bytecode itself is platform-independent, these Android-oriented pipelines are not directly transferable to iOS, where the native side is Objective-C/Swift compiled to ARM64 Mach-O rather than Java/DEX. We therefore choose Ghidra as the analysis framework because it supports native iOS binaries and can serve as a basis for future cross-language React Native analysis across platforms. We are not aware of a comparable public approach for framework-integrated Hermes analysis in Ghidra.

Lifting and IR-Based Analysis. Our work also builds on instruction-lifting research. SLED formalized instruction descriptions for retargetable tools [16], and Ghidra's SLEIGH language continues this line by translating instruction encodings and semantics into P-code. Tlyapova [18] provides a methodological precedent for specifying VM bytecode in Ghidra through a V8 bytecode processor module. Hermes differs because it is a register-based VM with many dynamically typed object and property operations.

4 Methodology

We implement HermesLoader as a Ghidra extension with three cooperating components: a loader that reconstructs Hermes bytecode containers as Ghidra program objects, a SLEIGH language specification for Hermes bytecode version 96, and an implementation of a PcodeInjectLibrary for Hermes-specific analyses. The main design goal is to make Hermes bytecode behave like an ordinary Ghidra analysis target without erasing the VM-level structure that is useful to analysts. Appendix A.1 gives representative implementation excerpts and details at the Ghidra API level.

4.1 Loader: Reconstructing Hermes Containers

The loader turns the compact Hermes bytecode file into a typed Ghidra memory image. It recognizes valid containers using the Hermes magic value and bytecode version, maps the header and section ranges into initialized memory blocks, and applies packed datatypes to major tables such as function entries, string entries, overflow strings, and debug metadata. These datatypes capture the physical layout of the bytecode file, including compact fields for function offsets, parameter counts, bytecode sizes, string offsets, UTF-16 flags, and short string lengths.

Datatype assignment alone is not sufficient, because many table fields encode values through bitfields or overflow records. After applying the raw structures, the loader reads the packed values back from memory, decodes them, and converts the results into a readable format. For example, string-table entries are decoded to recover the effective string offset, encoding mode, and length; the loader then creates references from descriptor entries to the corresponding string storage or overflow-string entries. Function-source metadata is treated similarly: raw identifiers are interpreted and re-expressed as references between functions and source strings. Debug metadata combines fixed-size datatype assignment with procedural decoding of variable-length SLEB128 source mapping records.

This reconstruction is important because SLEIGH alone sees only a byte stream. The loader creates the program objects that make lifted instructions navigable: function entry points receive Ghidra functions and labels, decoded strings are materialized as data, and metadata references are converted into ordinary cross-references. The loader also builds synthetic helper tables for strings, builtins, and loaded functions. These tables are not direct copies of Hermes metadata. Instead, they provide stable, analysis-friendly targets that hide compact encodings behind Ghidra addresses and references. For strings, the loader emits decoded string data, a uniform string table, and an additional helper table tailored to Ghidra's native cross-referencing and string-display behavior. Lifted string operands can then point to these helper entries rather than leaving the analyst to interpret raw string-table indices. For builtins, numeric Hermes runtime identifiers are mapped to synthetic callable targets that preserve names such as `JSON.parse` or `Math.abs`. For functions, the loader constructs a loaded-function table annotated with recovered symbolic names, allowing SLEIGH operands and later analyses to refer to function objects through stable memory-backed entries. These helper structures form an abstraction layer between Hermes's compact container format and the SLEIGH lifter. An initial design placed helper structures in a separate memory space, but Ghidra's cross-reference machinery did not resolve them reliably there; the current implementation places them at high, non-overlapping addresses in the default memory space.

4.2 Disassembling and Lifting Hermes Bytecode in SLEIGH

The SLEIGH specification defines Hermes instruction encodings, operands, and P-code translation [9, 10, 12]. Straightforward arithmetic, movement, comparison, branch and return instructions are mapped to conventional P-code operations where possible. However, many Hermes instructions operate on JavaScript-level concepts such as object properties, closures, arrays, builtins, and dynamically typed values. Lowering these instructions entirely into primitive operations would destroy semantic information that might be relevant to some reverse engineers or analyses. The trade-off we chose is to translate instructions relevant for control flow, comparisons and property names into their P-code equivalents, while preserving selected operations for *property access*, *aggregate construction*, and *dynamic dispatch* as named user-defined P-code operations, which are lifted to `CALLOTHER` operations.

Hermes uses a register-based virtual machine. Because SLEIGH cannot model an unbounded register file, we model the register set with bounded Ghidra register spaces: 256 value registers for the 8-bit operand form and a parallel view for long-register instructions. We also define named semantic registers for VM constants such as `True`, `False`, `Undefined`, `Null`, and `global`, together with synthetic call-interface registers for arguments, return values, return-site tracking, and error state. This keeps Hermes's register-oriented execution model visible while fitting Ghidra's processor abstraction.

Control-flow and call instructions require additional normalization. Branch targets are computed relative to the current instruction and exported as concrete program addresses, which lets Ghidra recover cross-references and control-flow graphs. Calls are normalized into a consistent convention: arguments are moved into

synthetic parameter registers, direct calls are resolved through the loaded-function table when possible, indirect calls retain their register-held targets, and results are read from a synthetic return register. Builtin calls receive their own dedicated user-defined operations: numeric Hermes builtin identifiers are mapped to named synthetic callees such as `Math.abs`, `JSON.parse`, `Object.create`, or `HermesInternal.*`. In the listing, these appear as ordinary calls with resolved targets, which fits Ghidra's call-analysis model while preserving the original Hermes builtin identity.

Handler-adjacent return and exception patterns are normalized into fused lifted forms to keep handler entry code visible to Ghidra. A more complete version should seed exception-handler targets directly from Hermes metadata during loading.

4.3 Decompiler Enrichment via P-Code Injection

Hermes-specific analyses are enabled through Ghidra's feature for injecting P-code, which an extension may ship via a PcodeInjectLibrary to refine the P-code emitted for individual instructions after the initial disassembly in SLEIGH. This feature is used for bytecode languages like Java where individual instructions benefit from being lifted while the whole program context is available. For Hermes, it creates a central design trade-off: named user-defined operations retain their Hermes-level identity for contextual enrichment, while their underlying VM semantics remain opaque to Ghidra's generic analyses.

Property access illustrates this choice. Hermes represents operations such as property reads through `getById` user operations rather than reducing them to primitive loads. The underlying instruction only encodes an index into the string table, but this table is not available to the SLEIGH lifter. Because the property access operation remains identifiable, the P-code injection library can use the full program context to rewrite this index as an address in the synthetic string segment. This address is then rendered as the property name by the Ghidra decompiler, which just sees a pointer to a string literal.

Keeping property access and dynamic dispatch operations identifiable also enables a backward analysis over Ghidra's P-code def-use graph, following copies, casts, indirect values, and selected merge nodes until we reach the originating property access operation. This recovers property access chains such as `obj.a.b.c()` or `global.JSON.stringify`. When such a chain resolves to the global object, the extension creates descriptive synthetic helper functions and redirects call references to them, making otherwise opaque dynamic dispatch sites visible in the call graph.

A second analysis handles assignment patterns. If a property is written exactly once through `putById` and later read through `getById`, we approximate the read as returning the previously written value. Repeated writes invalidate the approximation. This is useful for JavaScript patterns where a function is assigned to an object property and then invoked later, such as `o.m = f; o.m()`.

Aggregate construction follows the same strategy. By preserving its allocation and initialization operations, the analyses can identify the relevant sites, decode array and object literal buffers, annotate them with recovered keys and values, and inline string references where possible. The analyses also recover selected function references from metadata and closure-like initialization patterns. Small arrays can be expanded into visible constant elements, while larger aggregates are summarized through comments to avoid overwhelming the listing. Object literals are reconstructed from separate key and value buffers and attached to the corresponding allocation site, exposing structures such as request headers or configuration dictionaries. These enrichments do not attempt to reconstruct source-level JavaScript. Instead, they make the decompiler and listing views more useful by exposing names, constants, aggregate contents, and call targets that would otherwise remain hidden behind numeric table indices or opaque P-code user operations.

5 Evaluation

We evaluate HermesLoader along three dimensions: opcode coverage, agreement with the reference `hermesc` disassembler, and basic usefulness on real applications. These dimensions are complementary. Coverage measures whether the SLEIGH specification and loader can represent the Hermes instruction set at the opcode level. Disassembly agreement checks whether the recovered instruction streams match the upstream view of the same bytecode. The real-world check then asks whether the resulting Ghidra programs are stable enough to support ordinary reverse-engineering workflows on production applications.

5.1 Opcode Coverage

We built a JavaScript test suite in which each test function is designed to trigger a particular Hermes opcode. For example, a test for `NewObject` can simply return an object literal, causing the compiler to emit the corresponding bytecode instruction. We follow this pattern across the opcode set where possible, using small source snippets whose intended bytecode effect can be checked directly. A script compiles these tests with `hermesc`, dumps the resulting bytecode, and checks whether the expected opcode was emitted.

To increase coverage, the test suite is compiled under multiple optimization and feature configurations, because several instructions are emitted only under particular compiler modes. These include unoptimized and optimized builds as well as modes for static builtins, async break checks, basic-block profiling, block scoping, and WASM-related unsafe intrinsics. Additional filler inputs force large table indices and thereby exercise long-index encodings such as `GetByIdLong`. Appendix A.3 lists the configurations used.

The SLEIGH specification implements all 206 instructions in Hermes bytecode version 96. We count an instruction as implemented when its encoding is recognized, its operands are decoded, and its SLEIGH constructor emits either primitive P-code or a named P-code user operation preserving the corresponding Hermes VM operation. From ordinary JavaScript inputs and compiler configurations, our corpus generated 199 of these 206 instructions. The remaining seven instructions are effectively unreachable when compiling ordinary JavaScript source because they correspond to compiler-internal paths, dead backend instruction forms, unavailable non-enumerable property creation modes, or builtin-index ranges that current Hermes builds cannot exceed. Generating those instructions would therefore require modifying Hermes itself rather

than extending the JavaScript test suite. Appendix A.4 gives the opcode-level rationale.

5.2 Disassembly Agreement

We next loaded the generated bytecode files into Ghidra with HermesLoader and extracted the resulting disassembly for comparison against `hermesc -dump-bytecode`. For each generated function, we compare the recovered instruction sequence against the reference sequence with respect to instruction order, opcode identity, and operand values. We use `hermesc` as the reference because it is the upstream Hermes disassembler and therefore the closest available ground truth for opcode-level decoding. We do not use third-party Hermes reverse-engineering tools as reference outputs because they are independent implementations rather than authoritative specifications, and because HermesLoader targets navigable Ghidra program state rather than only textual disassembly.

This comparison validates instruction decoding and operand recovery, but it does not establish full semantic correctness of the generated P-code. We therefore treat disassembly agreement as a prerequisite for useful lifting rather than as a correctness argument.

Because the two tools render some equivalent constructs differently, we normalize decoded literals versus raw immediates, symbolic labels versus resolved addresses, builtin names versus synthetic call targets, and a small number of fused handler-adjacent instruction pairs. This removes presentation differences before comparing instruction order, opcode names, and operands.

Under this normalization, the remaining differences fall into a few recurring classes. Most are representational: Ghidra sometimes prints raw encoded operands where `hermesc` prints decoded literals or labels, and HermesLoader renders builtin calls as ordinary calls to resolved synthetic targets to fit Ghidra's call-analysis model. This affects, for example, constant-loading instructions, generator resume labels, branch targets, regular-expression operands, and builtin-call operands. The main non-representational limitation is `SwitchImm`: some jump-table successors are not yet seeded as code targets, so Ghidra can leave case bodies undisassembled. In those cases, `hermesc` reconstructs a fuller switch structure, while the current loader exposes the raw operands and recovers only part of the control flow encoded in the jump table. Appendix A.6 shows representative output pairs for these mismatch classes.

Across the corpus, we match the disassembly for the tested functions covering 199 of the 206 instructions in Hermes bytecode version 96. The remaining differences are concentrated in operand rendering and incomplete recovery of control-flow targets encoded in `SwitchImm` jump tables rather than basic opcode decoding.

5.3 Real-World Loadability Check

We did not perform a separate ground-truth evaluation of P-code semantics or decompiler correctness. Instead, we treat the absence of analysis failures and structurally incoherent output as supporting evidence for robustness, while acknowledging that this is an empirical usability check rather than a formal proof.

We therefore complement the synthetic corpus with a qualitative loadability and usability check on production applications. Following the collection approach of Magin et al. [6], we examined the top 2000 iOS applications and found 198 apps using the Hermes bytecode version supported by our loader. All 198 could be processed successfully: the loader recognized the container, created Ghidra functions for recovered Hermes function entries, generated P-code without analysis exceptions, and produced decompiler output for sampled functions. Manual inspection did not reveal obvious disassembly or decompiler errors.

Appendix A.2 illustrates the resulting analyst workflow. Searching for the recovered string `"Quick login failed:"` leads directly from the reconstructed string table to the relevant disassembly and decompiler output, allowing the analyst to inspect nearby error-handling behavior inside a real Hermes bundle. This check is not a formal semantic validation, but it provides evidence that the lifted representation is stable enough for practical interactive reverse engineering.

6 Discussion

Our results suggest that Hermes bytecode can be lifted into Ghidra in a way that is useful for interactive reverse engineering. The analyst-facing result comes from two layers. Disassembly and basic lifting make Ghidra's existing infrastructure applicable, including navigation, cross-references, control-flow recovery, and structured decompiler output, while our Hermes-specific enrichment passes add domain knowledge that Ghidra cannot infer on its own.

Three effects dominate the readability gain over textual bytecode disassembly. The first concerns control flow. Hermes conditional and unconditional jumps are lifted to ordinary branches, so Ghidra recovers basic blocks and function boundaries and the decompiler renders loops and conditionals as structured `if`/`while` constructs instead of the label-and-jump style of a linear bytecode listing. The second concerns string literals. Operands referring to the Hermes string table are resolved and inlined at their use sites, so property names, method names and message strings appear directly in the function body rather than as opaque string-table indices. The third concerns aggregate construction. The contents of array and object buffers are recovered and inlined, so a literal is visible as its initial values instead of an offset into the serialized buffer sections. The first effect follows from lifting alone, whereas the latter two, together with selected function references and synthetic helper targets, are contributed by the enrichment passes. The present evaluation does not isolate or quantify these contributions relative to the baseline decompiler output.

The design trade-off between primitive P-code and preserved user-defined operations is discussed in Section 4.3. Consequently, the decompiler output remains C-like pseudo-code over Hermes VM semantics rather than reconstructed JavaScript source, but the information that matters in practice is available, namely property names, method names, string constants, and aggregate literal contents. The decompiler primarily contributes by structuring the control flow based on the semantics of the control-flow and comparison instructions. This supports common reverse-engineering tasks on React Native applications, such as locating security-relevant functionality, tracing data flow into native modules and network calls, and recovering hard-coded secrets or endpoints. Because the decompiled output is plain structured text with resolved names,

it is also directly consumable by automated clients. Such decompiled output can further serve as input for LLM-based vulnerability analysis.

6.1 Limitations

The implementation and evaluation target Hermes 0.12.0 with Hermes bytecode version 96 and Ghidra 12.0.4, so other Hermes versions require updates to the loader and SLEIGH specification. Our opcode corpus is generated by compiling JavaScript test cases with `hermesc`, which makes it appropriate for decode-fidelity testing but also ties it to what `hermesc` can emit. The normalization used in the disassembly comparison could hide presentation defects, especially where raw immediates are accepted as equivalent to decoded literals. The real-world application check is qualitative and based on manual inspection, so it demonstrates practical robustness but not formal semantic correctness.

The differential comparison in Section 5.2 shows agreement on the decoded opcode stream across the corpus. Most residual differences are representational: HermesLoader renders encoded operand fields where `hermesc` renders resolved values (hexadecimal immediates and IEEE-754 bit patterns for constant loads, literal-table indices for BigInts and regular expressions, raw resume offsets for generator saves, absolute addresses instead of symbolic branch labels). These forms are semantically equivalent and can be eliminated by a post-load annotation pass that materializes the literal tables as typed data, attaches operand references to them, and emits equates for decoded constants and labels for resume points. Three differences are functional and do affect the recovered representation; Appendix A.6 gives the corresponding output pairs.

Jump-table control flow. The most substantive gap concerns `SwitchImm`. Both tools decode the instruction, but the loader does not follow the table-referenced successors, so the remaining case bodies stay classified as data. Affected functions therefore have incomplete control-flow graphs and bodies, cross-references into case bodies are missing, decompilation is degraded, and code reachable only through a switch table can be missed entirely. The instruction carries everything needed for static resolution (minimum and maximum case values, default target, table offset), so a dedicated analyzer can enumerate the entries, register them as a Ghidra `JumpTable`, force disassembly of each target, and type the table as data. Since `hermesc` already reports the complete target set, its output serves as ground truth for validating the recovered tables.

Builtin calls and fused forms. Rendering `CallBuiltin` as a resolved `Call` buys integration with call-target resolution, cross-referencing, and the decompiler, but erases the distinction between bytecode callees and VM-provided builtins and leaves the synthesized callee without a body or signature; materializing builtins as typed external or thunk functions in a dedicated namespace preserves both provenance and call-graph integration. Similarly, fused forms such as `RetAndCatch` make flow past a terminator explicit, but break the one-to-one correspondence between listing entries and Hermes instructions, complicating instruction-level diffing. Driving handler discovery from the per-function exception-handler tables and using explicit flow overrides would achieve the same effect without fusion.

Semantic soundness of the lifted P-code. Our differential comparison is performed at the disassembly level: it establishes that HermesLoader decodes the tested opcodes and operands consistently with `hermesc`, but not that the generated P-code is semantically equivalent to the bytecode in the Hermes VM. Divergence could arise from the modeling of individual instruction effects, including JavaScript coercions, side effects, and exceptional behavior. Named `CALLOTHER` operations present a different limitation: they deliberately preserve operations such as property access and dynamic dispatch without defining their complete effects in generic P-code. By contrast, enrichment rules such as propagating a uniquely observed `putById` value to a later `getById` are semantic approximations and may become unsound in the presence of aliases or prototype-chain behavior. Per-opcode property tests and differential execution of lifted P-code and Hermes bytecode on identical inputs could validate instruction results and state changes. The resulting P-code and decompiler output should therefore be treated as analysis aids rather than as a proven semantically equivalent reconstruction.

6.2 Future Work

Future work should extend support across Hermes bytecode versions. The loader should also recover exception-handler and switch-table targets directly from metadata to improve control-flow completeness. A more detailed evaluation of the enrichment passes could report before/after recovery counts for property access chains, indirect calls, aggregate literals, and string references on real applications. Additionally, integrating the Hermes representation with native iOS and Android code would enable broader cross-language analysis of React Native applications. Finally, the data-flow semantics through objects and environments are still represented by opaque `getById` calls in our lifter. While a straightforward lifting into field accesses as with JVM bytecode seems possible, this is not necessarily correct for a dynamic language like JavaScript, and needs to be evaluated as part of future work on data-flow analysis in Hermes via P-code.

7 Conclusion

We presented HermesLoader, a Ghidra extension that loads Hermes bytecode, lifts it to P-code, and reconstructs Hermes containers as typed program objects. By combining a SLEIGH specification, a companion loader, and Hermes-specific enrichment passes, the extension brings Hermes bytecode into Ghidra's standard disassembly, decompilation, navigation, and cross-reference workflows.

Our evaluation shows broad opcode-level coverage and agreement with the upstream `hermesc` disassembler: the lifter implements all 206 opcodes in Hermes bytecode version 96, our JavaScript test corpus generates 199 of them, and remaining disagreement themes are mostly representational apart from incomplete recovery of some control-flow targets. A complementary real-world sanity check on production bundles suggests that the recovered representation is already useful for practical analyst workflows. Overall, HermesLoader helps close a tooling gap for reverse engineering Hermes-based React Native applications and provides a foundation for future cross-language analysis in Ghidra.

Acknowledgments

This research work was supported by the National Research Center for Applied Cybersecurity ATHENE. ATHENE is funded jointly by the German Federal Ministry of Research, Technology and Space and the Hessian Ministry of Science and Research, Arts and Culture.

Part of the data used in this research was provided by the Appicaptor project[1].

References

[1] Lucas Baizer. 2026. Lucasbaizer/Hasmer. https://github.com/lucasbaizer/hasmer. (accessed 2026-06-15).
[2] Cifuentes, Cristina. 1994. *Reverse Compilation Techniques.* Ph. D. Dissertation. Queensland University of Technology, Brisbane.
[3] Cristian Falvo. 2023. *BN-Hermes : A Decompilation and Analysis Plugin for React Native's Hermes Bytecode.* Ph. D. Dissertation. Politecnico di Milano.
[4] Yonghui Liu, Xiao Chen, Pei Liu, John Grundy, Chunyang Chen, and Li Li. 2024. ReuNify: A Step Towards Whole Program Analysis for React Native Android Apps. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23).* IEEE Press, Echternach, Luxembourg, 1390–1402. doi:10.1109/ASE56229.2023.00113
[5] Yonghui Liu, Xiao Chen, Pei Liu, Jordan Samhi, John Grundy, Chunyang Chen, and Li Li. 2025. Demystifying React Native Android Apps for Static Analysis. *ACM Trans. Softw. Eng. Methodol.* 34, 4 (April 2025), 107:1–107:33. doi:10.1145/3702977
[6] Florian Magin, Fabian Scherf, Martin Renze, Cléo Fischer, and Gwendal Patat. 2025. Measuring While Playing Fair: An Empirical Analysis of Language and Framework Usage in the iOS App Store. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering (SURE '25).* Association for Computing Machinery, New York, NY, USA, 41–47. doi:10.1145/3733822.3764671
[7] Maritn Renze and Florian Magin. 2026. Fraunhofer-SIT/SURE2026-HermesLoader. https://github.com/Fraunhofer-SIT/SURE2026-HermesLoader. (accessed 2026-08-27).
[8] Meta Platforms, Inc. 2026. Facebook/Hermes. https://github.com/facebook/hermes. (accessed 2026-03-05).
[9] Meta Platforms, Inc. 2026. Hermes/Doc/Design.Md at Main · Facebook/Hermes. https://github.com/facebook/hermes/blob/main/doc/Design.md. (accessed 2026-03-05).
[10] Meta Platforms, Inc. 2026. Hermes/Doc/VM.Md at Main · Facebook/Hermes. https://github.com/facebook/hermes/blob/main/doc/VM.md. (accessed 2026-03-05).
[11] Meta Platforms, Inc. 2026. Using Hermes · React Native. https://reactnative.dev/docs/hermes. (accessed 2026-03-05).
[12] National Security Agency. 2026. GhidraDocs/Languages/Html/Sleigh.Html · NationalSecurityAgency/Ghidra. https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/languages/html/sleigh.html. (accessed 2026-03-05).
[13] National Security Agency. 2026. NationalSecurityAgency/Ghidra. https://github.com/NationalSecurityAgency/ghidra. (accessed 2026-03-05).
[14] P1sec. 2026. P1sec/Hermes-Dec. https://github.com/P1sec/hermes-dec. (accessed 2026-06-15).
[15] Pilfer. 2026. Pilfer/Hermes_rs. https://github.com/Pilfer/hermes_rs. (accessed 2026-06-15).
[16] Norman Ramsey and Mary F. Fernández. 1997. Specifying Representations of Machine Instructions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (May 1997), 492–524. doi:10.1145/256167.256225
[17] Pongsakorn Sommalai. 2026. Bongtrop/Hbctool. https://github.com/bongtrop/hbctool. (accessed 2026-06-15).
[18] Natalya Tlyapova. 2021. Creating a Ghidra Processor Module in SLEIGH Using V8 Bytecode as an Example. https://swarm.ptsecurity.com/creating-a-ghidra-processor-module-in-sleigh-using-v8-bytecode-as-an-example/. (accessed 2026-06-15).

Generative AI Usage

GPT-5.4 was used to improve the grammar and spelling of the paper, and to assist in creating both the Python scripts and the JavaScript code (compiled to Hermes bytecode) used in the evaluation. All AI-generated code was manually reviewed and validated by the authors.

[1]https://www.appicaptor.com

A Appendix

A.1 Representative Methodology Implementation Details

This appendix subsection collects representative implementation details that are omitted from the main methodology discussion to keep that section focused on design ideas rather than engineering mechanics.

Representative SLEIGH Excerpts. The following snippets illustrate the style of the SLEIGH implementation used to encode high-level Hermes operations, conditional control flow, register definitions, normalized calls, builtin resolution, split return/exception behavior, and helper-table address materialization.

```
define pcodeop addEmptyString;

:AddEmptyString val_r0_Reg8, val_r1_Reg8 is op=0x80;
 val_r0_Reg8; val_r1_Reg8 {
val_r0_Reg8 = addEmptyString(val_r1_Reg8);
}
:JmpTrue dest, val_r0_Reg8 is op=0x90;
 dest; val_r0_Reg8 {
if (val_r0_Reg8 == True) goto dest;
}
dest: rel is p0_Addr8[rel = inst_start + p0_Addr8;] {
export *[ram]:4 rel;
}
define register offset=0xC0000 size=4
 [ True False Undefined Null global ];
define register offset=0xA0000 size=4
 [ return_address_Reg8 ];

define register offset=0xB0000 size=4 [
param0_Reg8 param1_Reg8 param2_Reg8 param3_Reg8
param4_Reg8 param5_Reg8 param6_Reg8 param7_Reg8
param8_Reg8 param9_Reg8 return_Reg8 error_Reg8
];
:Call1 val_r0_Reg8, val_r1_Reg8, val_r2_Reg8 is
op=0x51; val_r0_Reg8; val_r1_Reg8; val_r2_Reg8 {
param0_Reg8 = val_r2_Reg8;
return_address_Reg8 = inst_next;
call [val_r1_Reg8];
val_r0_Reg8 = return_Reg8;
}
:CallBuiltin val_r0_Reg8, 0x01, p1_UInt8 is op=0x59;
 val_r0_Reg8; p0_UInt8=0x01; p1_UInt8{
val_r0_Reg8 = Date.UTC(p1_UInt8:1);
}
:RetAndCatch val_r0_Reg8, val_r1_Reg8 is op=0x5c;
 val_r0_Reg8; op2=0x5d; val_r1_Reg8 {
return_Reg8 = val_r0_Reg8;
if (error_Reg8 != 0) goto inst_next;
return [return_address_Reg8];
catch(val_r1_Reg8);
}
```

```
destString: rel is p0_StringIDUInt32[rel =
 p0_StringIDUInt32 * 9 + 0xA0000000;] {
local stringTableAddress:4 = *:4 rel:4;
export stringTableAddress;
}
destFunction: rel is p0_FunctionIDUInt16[rel =
 p0_FunctionIDUInt16 * 8 + 0xF0000000;] {
local functionTableAddress:4 = *:4 rel:4;
export functionTableAddress;
}
```

Loader Engineering Details. At the implementation level, the loader materializes container sections using `mem.createInitializedBlock(name,...)`. Packed metadata layouts are represented through Ghidra `StructureDataType` definitions and then instantiated as typed `Data` objects over the recovered table ranges. Representative concrete layouts include 16-byte function-table spacing, 4-byte packed string-table spacing, 9-byte synthetic string-table entries, and 8-byte loaded-function entries.

Analysis Pass Engineering Details. Several analysis passes also depend on specific Ghidra APIs and internal representations. Representative examples include `DecompInterface`, `HighFunction`, and `PcodeOpAST` for backward property access recovery; `Memory.createInitializedBlock`, `CreateFunctionCmd`, `DisassembleCommand`, and `ReferenceManager.addMemoryReference` for synthetic helper functions; and `FlatProgramAPI.setPreComment`, `DataUtilities.createData`, and `ParallelDecompiler` for aggregate recovery and function-reference recovery.

A.2 Real-World Analysis Screenshots

The following screenshots show representative Ghidra views produced by HermesLoader when loading and inspecting a real-world Hermes application.

A.3 Evaluation Configurations

Opcode-generation and disassembly experiments used a local Debug build of `hermesc` from Hermes release 0.12.0, targeting Hermes bytecode version 96, together with Ghidra 12.0.4. The JavaScript opcode-generation suite was compiled under multiple Hermes configurations, including `-O0`, `-O`, `-fstatic-builtins`, `-emit-async-break-check`, `-basic-block-profiling`, and `-block-scoping`. Specialized filler-based variants were used to force long-index encodings. WASM-specific opcodes additionally required `-funsafe-intrinsics` and a Hermes build with `HERMES_RUN_WASM=ON`.

A.4 Source-Unreachable Opcodes

Seven implemented Hermes opcodes were excluded from source-based generation because they are effectively unreachable in the current compiler implementation. `Unreachable` and `CoerceThisNS` are compiler-internal artifacts rather than stable outcomes of ordinary JavaScript compilation: `Unreachable` is introduced only along a debug-only internal path, while `CoerceThisNS` is created transiently during optimization and simplified away before bytecode emission. `CallDirect` and `CallDirectLongIndex` belong to an instruction family that remains in the backend but is no longer emitted by active compilation passes. `PutNewOwnNEById` and `PutNewOwnNE`

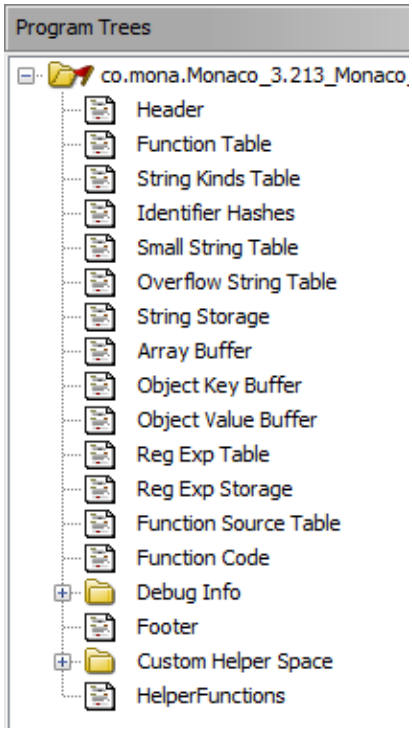


Figure 1: Recovered Hermes bytecode container regions materialized as Ghidra program-tree entries.

`ByIdLong` require a non-enumerable property-creation mode for which the enum value exists, but current creation sites hardcode the enumerable variant. Finally, `CallBuiltinLong` is theoretically valid but practically unreachable because it requires a builtin index beyond the short-form encoding range, whereas Hermes exposes far fewer builtins than that threshold.

A.5 Example: Decompiled Network Request

To illustrate the analyst-facing value of recovered constants, property access chains, aggregate literals, and call targets, we also analyzed a controlled React Native example that sends GPS-derived location data through an HTTP request. The original JavaScript constructs a request body from latitude, longitude, and timestamp properties, serializes it with `JSON.stringify`, and sends it using `fetch`. The corresponding HermesLoader output preserves the relevant property access chains, aggregate construction steps, request headers, request body, abort signal, and network endpoint. Listings 1 and 2 show the comparison. This example is not intended as a semantic equivalence proof; rather, it shows the analyst-facing form in which privacy-relevant application behavior can be inspected after lifting.

A.6 Representative Disassembly Output Pairs

Reference examples for the difference classes discussed in Section 6.1.

SaveGenerator / SaveGeneratorLong. `SaveGenerator` is decoded correctly by Ghidra at the opcode level, but Ghidra renders the

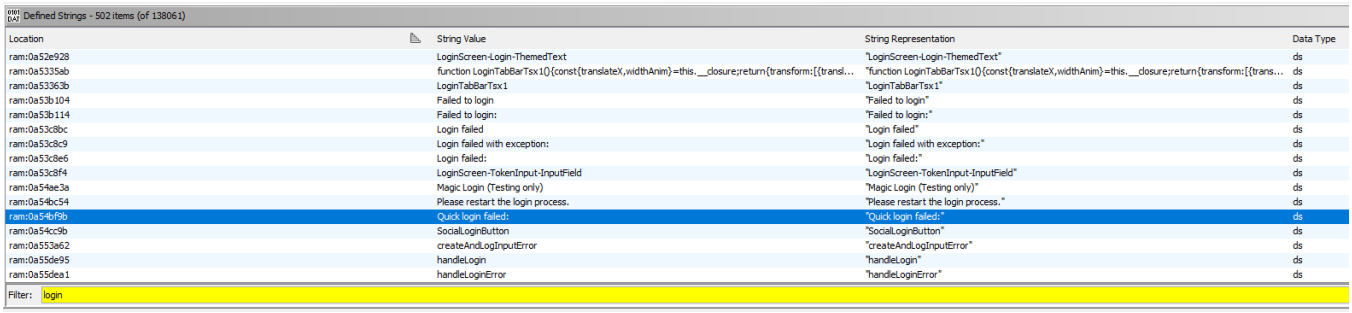


Figure 2: Filtering the recovered string table for login-related strings in a real-world Hermes application.

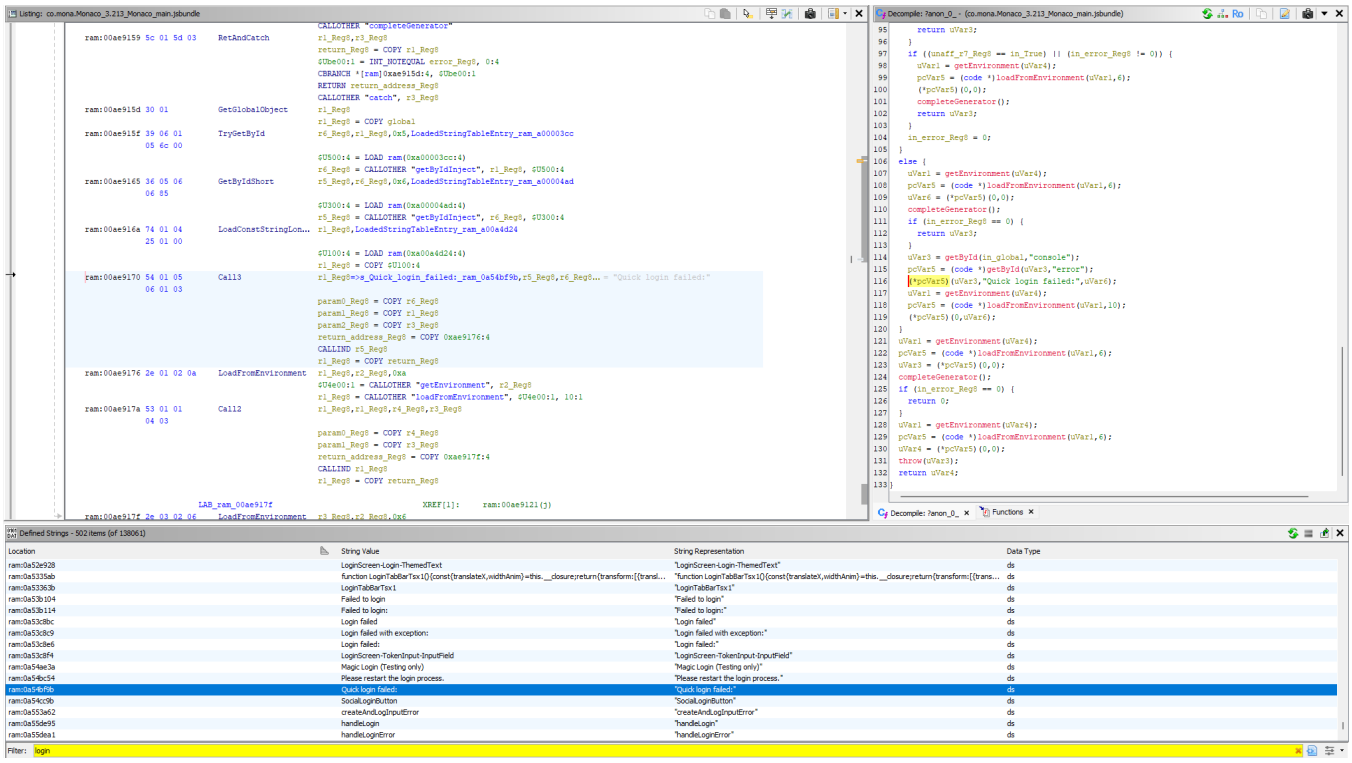


Figure 3: String-guided navigation from the recovered string table to the corresponding disassembly and decompiler output in a real-world Hermes application.

operand as a raw encoded resume offset, whereas `hermesc` resolves it to a symbolic resume label.

```
hermesc: SaveGenerator L2
Ghidra: SaveGenerator 0x4

hermesc: SaveGeneratorLong L3
Ghidra: SaveGeneratorLong 0x8a
```

LoadConstBigInt, LoadConstBigIntLongIndex, CreateRegExp. For these opcodes, `hermesc` prints the referenced literal contents directly, while HermesLoader/Ghidra prints only the corresponding literal-table index or reference. These differences are representational and do not indicate a semantic disagreement in the decoded instruction stream.

```
hermesc: LoadConstBigInt r1, 9007199254740991n
Ghidra: LoadConstBigInt r1_Reg8, 0x0

hermesc: LoadConstBigIntLongIndex r1,
              9007199254740991n
Ghidra: LoadConstBigIntLongIndex r1_Reg8, 0x10004

hermesc: CreateRegExp r1, "hello\s+world", "gi", 0
Ghidra: CreateRegExp r1_Reg8, 0x15f, 0x151, 0x0
```

Listing 1: Decompiler output recovered from the Hermes bytecode.

```
uVar11 = newObject();
uVar1 = getEnvironment(uVar4);
uVar9 = loadFromEnvironment(uVar1,0);
uVar12 = getById(uVar9,"coords");
uVar12 = getById(uVar12,"latitude");
puVar2 = (undefined4 *)putNewOwnById(uVar11,"latitude");
*puVar2 = uVar12;
uVar12 = getById(uVar9,"coords");
uVar12 = getById(uVar12,"longitude");
puVar2 = (undefined4 *)putNewOwnById(uVar11,"longitude");
*puVar2 = uVar12;
uVar9 = getById(uVar9,"timestamp");
puVar2 = (undefined4 *)putNewOwnById(uVar11,"timestamp");
*puVar2 = uVar9;
iVar5 = newObject();
puVar2 = (undefined4 *)putNewOwnById(iVar5,"method");
*puVar2 = "POST";
uVar9 = newObject();
puVar2 = (undefined4 *)putNewOwnById(uVar9,"Content-Type");
*puVar2 = "application/json";
puVar2 = (undefined4 *)putNewOwnById(iVar5,"headers");
*puVar2 = uVar9;
uVar9 = getById(in_global,"JSON");
uVar9 = global.JSON.stringify(uVar9,uVar11);
puVar2 = (undefined4 *)putNewOwnById(iVar5,"body");
*puVar2 = uVar9;
uVar9 = getById(uVar10,"signal");
puVar2 = (undefined4 *)putNewOwnById(iVar5,"signal");
*puVar2 = uVar9;
pcVar3 = (char *)global.fetch(0,"http://example.test",iVar5);
```

Listing 2: Original React Native source for the controlled example.

```
const body = {
        latitude: location.coords.latitude,
        longitude: location.coords.longitude,
        timestamp: location.timestamp,
};

const response = await fetch("http://example.test", {
        method: 'POST',
        headers: {
                'Content-Type': 'application/json',
        },
        body: JSON.stringify(body),
        signal: controller.signal,
});
```

LoadConstInt and LoadConstDouble. For constant-loading instructions, `hermesc` prints the decoded constant value, whereas HermesLoader/Ghidra prints the encoded immediate representation. For integers, this appears as hexadecimal rather than decimal; for doubles, this appears as the raw IEEE-754 bit pattern rather than the rendered floating-point literal.

```
hermesc: LoadConstInt r1, 100000
Ghidra: LoadConstInt r1_Reg8, 0x186a0

hermesc: LoadConstDouble r1, 3.14159
Ghidra: LoadConstDouble r1_Reg8, 0x400921f9f01b866e
```

CallBuiltin. For calls that `hermesc` renders as `CallBuiltin`, HermesLoader instead displays a regular `Call` with a resolved callee target and an explicit argument-count operand.

```
hermesc: CallBuiltin r0, "Math.abs", 2
Ghidra:  Call r0_Reg8, global.Math.abs, 0x2

hermesc: CallBuiltin r7,
            "HermesBuiltin.ensureObject", 3
Ghidra:  Call r7_Reg8,
            global.HermesBuiltin.ensureObject, 0x3
```

Jump Instructions. For both the normal and long jump variants, the two disassemblers agree on the opcode semantics, but differ in how branch targets are rendered. `hermesc` prints symbolic labels, whereas Ghidra prints resolved absolute target addresses. This affects `Jmp`, `JmpTrue`, and `JmpFalse` as well as their long-form counterparts.

```
hermesc: Jmp L2
Ghidra:  Jmp 0x0000a5dd

hermesc: JmpTrue L2, r1
Ghidra:  JmpTrue 0x0000ae7a, r1_Reg8

hermesc: JmpFalse L1, r1
Ghidra:  JmpFalse 0x0000aebe, r1_Reg8

hermesc: JmpLong L2
Ghidra:  JmpLong 0x0000ae60

hermesc: JmpTrueLong L2, r1
Ghidra:  JmpTrueLong 0x0000af4f, r1_Reg8

hermesc: JmpFalseLong L1, r1
Ghidra:  JmpFalseLong 0x0000b125, r1_Reg8
```

SwitchImm. The `SwitchImm` instruction is recognized correctly by both `hermesc` and Ghidra. However, `hermesc` reconstructs the full switch semantics, including symbolic labels and the complete jump-table target set, whereas Ghidra exposes only the raw encoded operands and does not disassemble all table-referenced successor blocks. As a result, the remaining case bodies are treated as data rather than code.

```
hermesc: SwitchImm r0, 84, L11, 0, 9
Ghidra: SwitchImm r0_Reg8, 0x54, 0x4e, 0x0, 0x9
```

Fused Instructions. Ghidra emits several fused instructions, while `hermesc` emits equivalent unfused opcode sequences.

```
Ghidra: ThrowAndCatch r1_Reg8, r1_Reg8
hermesc: Throw r1
         Catch r1

Ghidra: RetAndCatch r8_Reg8, r10_Reg8
hermesc: Ret r8
         Catch r10

Ghidra: RetAndResumeGenerator r5_Reg8, r2_Reg8, r8_Reg8
hermesc: Ret r5
         ResumeGenerator r2, r8
```