BiNit: Nitpicking Semantic Bugs in x64 Binary Lifters

Jack Royer
CentraleSupélec
Rennes, France
Thales
Rennes, France
jack.royer@centralesupelec.fr

Frédéric Tronel
IRISA
CentraleSupélec, Inria, CNRS,
Université de Rennes
Rennes, France
frederic.tronel@centralesupelec.fr

Yaëlle Vinçont
IRISA
Université de Rennes, Inria, CNRS
Rennes, France
yaelle.vincont@ens-rennes.fr

Abstract

The first steps to analyzing a binary file, especially when using a binary analysis tool, are usually disassembly followed by lifting to an intermediate representation. This translation, from the bytes to a more expressive language, is a crucial part of the process, as it allows humans to reason about the program and tools to analyze it. As a result, any inaccuracy in the lifting can compromise the analysis as a whole.

However, to test a binary lifter, one needs to understand the assembly language it lifts from. This can be challenging due to complex semantics, undefined behavior, and a lack of ground truth. Moreover, since each tool often comes with its own lifter and IR, an efficient tester needs to be easily reusable across tools.

We introduce BiNit, a framework for testing x64 binary lifters. BiNit compares concrete values from instruction executions and their emulated lifted counterparts, making lifter bugs easier to detect and diagnose. We ran BiNit on 7 state-of-the-art tools and identified 35 semantic bugs, showing that semantic lifting errors remain common even in mature binary analysis frameworks.

CCS Concepts

• **Software and its engineering → Software reverse engineering**; *Software verification and validation.*

Keywords

binary lifting, lifter testing, binary analysis, x64, instruction semantics

ACM Reference Format:
Jack Royer, Frédéric Tronel, and Yaëlle Vinçont. 2026. BiNit: Nitpicking Semantic Bugs in x64 Binary Lifters. In *2nd Workshop on Software Understanding and Reverse Engineering (SURE '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3847158.3847383

1 Introduction

In the realm of binary analysis, few analyses are performed directly on the binary. Instead, binaries are often translated (or "*lifted*") into an Intermediate Representation (IR), and analyses are then run on the translated code. Analyzing an IR – as opposed to machine code – offers several advantages, such as being architecture-independent, working with a reduced instruction set, and tailoring the IR to simplify the intended analyses.

However, working with an IR has a major drawback: translation errors. If you are analyzing the IR of a program, for example, to detect runtime errors statically, you are assuming that the translation is semantically equivalent to the binary program. Translation errors undermine this assumption, potentially leading to errors during analysis passes and inaccurate results.

As with other programs, testing lifters helps identify those errors. Unlike other programs, testing lifters requires a specific framework to determine the ground truth for each instruction's results, as there are too many to specify an oracle for each. Moreover, an oracle derived for a given assembly language could be used to test any lifters working on this assembly language, provided that integration of a new lifter into the framework is easy.

In this paper, we focus on the widely used Intel x64 architecture. We describe our testing framework, BiNit. BiNit tests instructions individually, allowing for fine-grained identification of translation errors. When a test fails, it is immediately clear which instruction is at fault, making debugging and fixing the lifter much easier. Moreover, testing individual instructions allows us to generate a large number of tests. BiNit stores these tests and expected values in a database, making integration with different lifters simple, regardless of the lifter's IR or implementation language.

Our contributions are as follows:

- We present BiNit, an open-source framework for testing the accuracy of lifters by executing individual instructions and comparing the resulting architectural state with that produced by emulating the lifted semantics.
- We introduce and release[1] a database of these instruction-level test cases as an artifact, containing concrete instruction instances, input CPU states, and the corresponding output states observed on real hardware.
- We evaluate BiNit on 7 widely used binary analysis tools and found 35 bugs, demonstrating the practical value of BiNit and showing that instruction-level testing can expose semantic discrepancies.

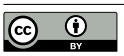
We reported the discovered bugs upstream with detailed human-written reports, precise reproduction steps, and actionable explanations of the underlying semantic mismatches.

2 Problem Statement

2.1 Lifter contract

Binary analysis relies on a lifter being *correct*. One might assume that a *correct* lifter is one that perfectly captures the semantics of

[1] BiNit is available at https://binit.wazab.in



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/10.1145/3847158.3847383

every instruction available. However, depending on the use cases, *correctness* can vary. For instance, an IR may only need to support CFG (Control-Flow Graph) recovery, lightweight data-flow analysis, or faithful instruction emulation. And one might only want these features on a subset of instructions. Not all lifters are designed to capture the semantics of every instruction perfectly. We define a *lifter's contract* as the set of architectural behaviors that the lifter claims, explicitly or implicitly, to model for a given use case.

For example, `radare2`'s IR, called ESIL, is explicitly defined as being partial: *"Radare2 employs its own intermediate language and virtual machine, known as ESIL, for partial emulation (or imprecise full emulation)"* [31]. This makes sense, as ESIL is meant to be human-readable to allow analysts to *"see what a program does while it is running, even for the most cryptic architectures"* [31].

With that in mind, other use cases, such as deobfuscation, require extremely precise lifting. Indeed, attackers can exploit bugs in lifters and use discrepancies to determine if they are being analyzed or emulated. Some malware goes so far as to use undefined flag values to evade emulation [39]. To effectively analyze such binaries, we need to be highly confident in our lifter's accuracy.

Thus, lifter correctness is best understood relative to a *contract*: which instructions, architectural state components, and side effects the lifter is expected to model. Accordingly, in our evaluation, we do not report or count a discrepancy as a bug when it falls outside the evaluated lifter's contract.

2.2 Challenges of Testing Intel x64 Lifters

Testing Intel x64 lifters is challenging for several reasons:

Semantic sensitivity. Small changes during lifting can have a large impact downstream. If the lifter improperly translates an instruction, this can quickly propagate throughout the binary, corrupting semantics on a wider scale. This means that tests targeting multiple instructions make it difficult to isolate the source of the discrepancy.

State space explosion. Instructions often read and/or write multiple values to registers, immediate values, memory, and flags. A *perfect* lifter should capture all of these semantics. Therefore, all these side effects need to be tested.

This leads to a new problem: state explosion. Ideally, one would want to verify every instruction for every possible input value.

However, combining Intel's large number of instructions[2] with every register, immediate, and memory value possible leads to a colossal amount of test cases. Exhaustive validation is impossible; instead, a tester must rely on fuzzing or diffing.

No simple ground truth. The core problem with testing an x64 lifter is the lack of a ground truth. Although a lifter should describe the CPU's semantics, they are somewhat opaque. Though a formal specification exists for certain architectures [20], none exists for the Intel x64 architecture. To make matters worse, Intel's roughly 5,000-page reference manual [23] is notoriously difficult to understand and not even trustworthy [21]. The pseudocode in the manual is merely suggestive and cannot be easily automated.

This makes it an especially appropriate target for our technique, since we do not require a *formal specification*, given that we use the processor's behavior as the ground truth.

Arguably, even if a true specification existed, a *perfect* lifter would describe the real behavior of a program on a physical CPU rather than the theoretical behavior of the specification. This matters when the CPU and the specification disagree, either due to a bug or to undefined behavior.

Undefined Behavior. In Intel's specification, certain values are undefined. For example, the Auxiliary Flag (`AF`) has an undefined value after a `XOR` instruction. It's important to note that undefined does not mean the value is not readable. It also does not mean that the value is non-deterministic. This is a serious issue for lifters as attackers exploit this behavior [39].

For testers, this is also an issue as undefined values often lie outside a lifter's *contract*. If our tester is too strict on undefined values, it will lead to many false positives.

2.3 IR mismatch and diversity

The final problem is the diversity of IRs. Binary analysis is a notoriously fragmented field with many different tools, lifters, and IRs. To illustrate this point, we have compiled a list of all IRs and lifters we know of in Table 2 (in Appendix).

This diversity of lifters matters to us because an effective tester needs to test multiple tools easily. Indeed, most of the tools in the table have their own lifter, which may have bugs and need testing. A key metric for a successful tester is therefore how easily it can be used across lifters.

3 Methodology

We developed BiNit, a framework to test the accuracy of x64 lifters.

3.1 Scope

With this project, we target semantic errors in Intel x64 lifters. We built our scope based on our analysis of lifters and their contracts. For example, privileged instructions rarely appear in the contracts, making testing them unproductive. Similarly, we were initially interested in testing `MMX`, `SSE`, `AVX`, and `AVX-512`[3] instructions, however, as they fall outside of most contracts, they were omitted. In the end, we consider "in-scope" instructions matching all of the following criteria:

- known, i.e. in the Intel Manual [23] – no undocumented instructions ;
- unprivileged ;
- arithmetic – no floating-point or vector instructions.

Additional instructions may be added at a later date, but we found those sufficient for our targeted lifters.

3.2 Core idea

BiNit relies on an extremely simple strategy. We start by generating *"interesting"* (as defined in 3.3.1) CPU states for a list of instructions.

[2]When counting all `ICLASS` in XED[24], we find 1,961 instructions. This is counting all prefixes, suffixes, extensions, and legacy instructions. Narrowing the search to instructions from `BASE`, `LONGMODE`, `ADOX_ADCX`, `BMI1`, `BMI2` extensions and excluding prefixes, suffixes and instructions labeled `INVALID`, we get 245 instructions.

[3]Intel SIMD/vector instruction-set extensions.

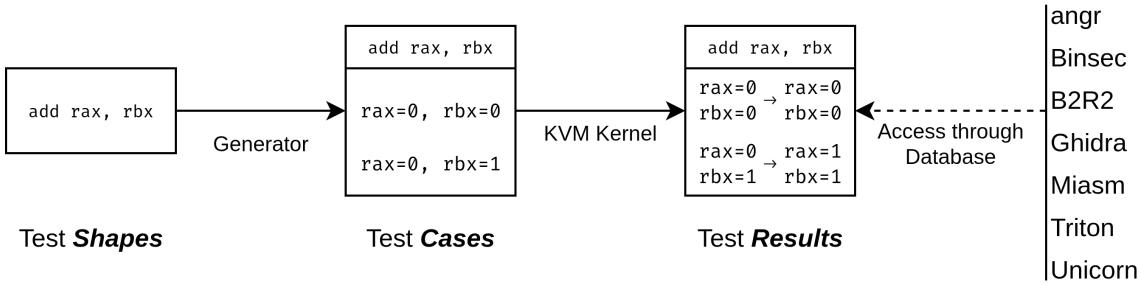


Figure 1: BiNit Test Architecture

For each (`CPU state, instruction`) pair, we execute the instruction on a physical CPU to yield our ground truth. This ensures perfect accuracy.

We then lift that same instruction and emulate the corresponding IR using the same initial CPU state. Finally, we compare the final emulated CPU state to the physical one.

This strategy assumes very little from the tested lifter or its IR. Moreover, to facilitate integration with various IRs, we store the ground truth in a database. The database acts as the interface between our ground truth and lifters.

BiNit's architecture makes integration with various lifters, programmed in various languages, very simple. Lifters do not need to depend on a custom library; they can connect to our database (see Appendix A) using any off-the-shelf SQL library.

3.3 Architecture

In Section 3.1, we described the set of instructions currently implemented. However, we want to ensure that our infrastructure can be easily adapted to test additional instructions.

- We want to be able to execute any (including privileged) instruction in a controlled environment;
- We want to be able to accurately capture the entire CPU state immediately after executing a single instruction;
- We want our test bed to be as fast as possible to test a very large part of the instruction set;
- We want to capture exceptions raised by the CPU.

3.3.1 Tests generation. We created a large set of test *shapes*, using an LLM. This resulted in a 1K-line JSON file. Each *shape* specifies an instruction, as well as the operand form we want to test. For example, the following patterns would correspond to test *shapes*: "`ADD r/m64, r64`" ; "`XOR r/m16, imm16`". Moreover, for each instruction, we can specify additional metadata, such as implicit register reads/writes or which flags are undefined.

These test *shapes* are then passed through a Python generator to fabricate both an actual instruction and a set of *interesting* CPU input states, creating a test *case*. The instruction is generated by replacing patterns with actual values (e.g. `r/m64` with `RAX`). Then the state is generated by setting all registers to *interesting* values such as `0x01`, `0xFFFFFF01`, or `0xFFFFFFFFFFFFFFFF`[4]. We want to assign values to registers that will be read and written by the instruction so that we can detect any modifications. For example, if an instruction is supposed to write the value 1 in `RAX`, this will only be visible if `RAX`'s initial value isn't 1.

To handle memory operations, we added a phony register `mem value` that represents the value at a constant memory address. Test *cases* can therefore specify a memory value in the same way that they specify a register value.

In total, we generated 1,718,948 test *cases* covering 198 instructions. This covers 80% of the in-scope instructions listed by XED[24]. This generation step is summarized in Figure 1.

The remaining instructions either did not work in our model (`hlt`, `ud`), would require a significantly more complex harness (`cpuid`, `in`), or were only encodable in 32 bits (`lds`, `les`).

We also experimented with random generation, but it did not achieve the desired coverage.

3.3.2 Client-Kernel architecture. Our approach for inspecting the CPU states is heavily inspired by LibLisa [13]. The core idea is to place instructions at the end of a page, with the next page (a guard page) being non-present. This way, a page fault is triggered immediately after the instruction is executed. By installing a custom page fault handler, we can capture the CPU state immediately after an instruction executes. This also guarantees high fidelity, as we are certain that no other instructions altered the state. For instructions that affect the control flow, we ensure they also land on the guard page.

To implement this, we built a custom micro kernel in Rust (1K lines of code). This kernel runs inside Qemu, using Kvm to ensure that instructions are executed natively on the host CPU. We implemented a Rust client that runs on the host and communicates with the kernel. The bottleneck in BiNit and any similar framework is communication between the host and the kernel. We chose to rely on Qemu's Shm device, which provides a shared memory region between the host and the guest and is the fastest solution in our experiments.

An overview of the architecture is shown in Figure 2 and described below.

(1) The client fetches a test case and sends it to the kernel;
(2) The orchestrator sets up the test by writing the opcode, setting the context, and jumping (using `iretq`[5]) to the instruction;
(3) An *instruction fetch page fault* is thrown after the execution of the instruction;

[4] taken from QEMU's testing code [4]

[5] `iretq` returns from an instruction by restoring the saved context.

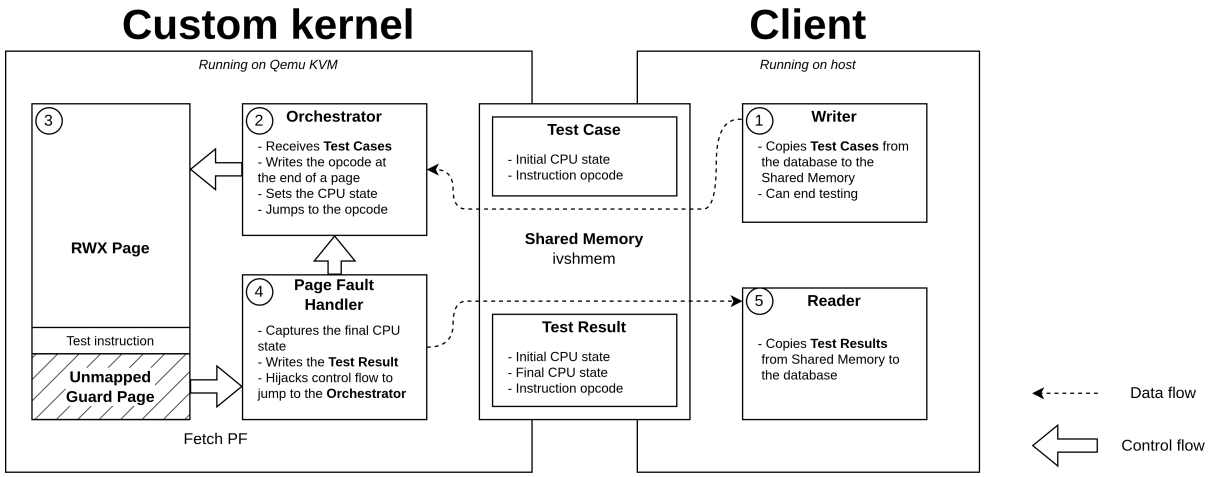


Figure 2: BiNit Ground Truth Generator Architecture

(4) The custom handler catches the instruction, captures the CPU state, and sends it to the client. It also modifies the stack frame so that `iretq` jumps back to the orchestrator;
(5) The client-side reader retrieves the test result and can use it.

Being in a custom kernel gives us full control over the execution environment. We can execute privileged instructions, fully set the CPU state before executing an instruction, and capture the entire CPU state after execution.

4 Evaluation

In this section, we differentiate between:

warnings raised by BiNit when the behavior of an instruction differs from the ground truth;
errors found in the targeted lifters, through manual investigation of a warning;
bugs in our implementation.

4.1 Tools

We decided to test BiNit on a set of popular binary analysis tools, written in different languages and using different IRs and lifters. This choice ensures our tool is easily integrable. For example, in Python-based IRs, the integration was done in around 200 unique lines of code (193 for `angr`, 202 for miasm, 175 for Triton, 138 for Unicorn) using the generic `psycopg` library. LLMs completely generated these harnesses. We deliberately did not target commercial closed-source tools such as Binary Ninja[5] or IDA Pro[22]. Indeed, since our primary goal is to enable a developer to evaluate their own tool, we require access to the lifters' source code to differentiate bugs in our tool from genuine lifting errors. However, we encourage these vendors to test their lifter using BiNit.

Disclosure Process. Although the bugs are unlikely to impact system security significantly, they could still place an unwanted strain on open-source developers. For each bug we found, we manually analyzed it, reproduced it, and attempted to find a fix.

Tools in the dataset (alphabetical order). We briefly present the tools in our datasets. Target versions are indicated in Table 1.

`angr` [35] is a binary analysis platform written in Python. It provides both static and dynamic symbolic analysis. These analyses leverage Valgrind's VEX IR [6].

Binsec [15] is a binary analysis framework written in OCaml. It uses its own DBA IR. However, although the project is open-source, the lifter (based on a tool called unisim) is not: it is only available as a pre-compiled binary. As such, we don't know much about the lifter contract or test cases.

B2R2 [25] is written in F# with a focus on lifting speed and describes itself as a collection of open-source libraries and tools. It uses its own IR: LowUIR. B2R2 contains a very large amount (300) of tests for x64 decompilation, but only two for lifting. In these two tests, they compare the IR for the add instruction to a given string.

Ghidra [28] is probably the most popular open source binary analysis tool. Ghidra uses the PCode IR. The lifter is built using the SLEIGH DSL, which handles both disassembly and lifting. In our tests, we used a modified version of Ghidra's SLEIGH specifications[7] that included community fixes and PRs not yet merged into Ghidra's official repository. This allowed us to identify new bugs rather than rediscover existing ones. For example, the `xadd` instruction bug was found and already patched by Chesser for his project icicle [10]. Furthermore, we used our own SLEIGH compiler and emulator to test the SLEIGH specifications independently of Ghidra's SLEIGH infrastructure. Our implementation did, of course, contain bugs; however, we were able to patch our implementation and focus on SLEIGH bugs.
Ghidra has very few tests. It emulates small C functions and compares the results to known values.

[6] More precisely, a custom Python version of VEX IR.
[7] available here: https://github.com/Garfield1002/open_sleigh

Table 1: Errors found in tested tools

Tool Version	Semantic Lang	Test Lang	Flag Errors	Register Errors	Disassembly Errors	Ignored Warnings
Unicorn 2.1.4	C	Python	None	`ret`, `rorx`, `shld` `shrd`, `pdep`, `blsi` `bzhi`, `cmpxchg`	None	11
Miasm 0.1.5	Python	Python	None	None	`popcnt`, `nop ebx` `adcx`, `adox`	44
Binsec 0.11.1	OCaml	OCaml	(`AF`†) `popcnt (CF)` `imul`, `sar`, `shrd`	None	`popcnt ax, bx` `movzx rax, bx` `crc32 eax, bl`	28
B2R2 0.9.1	F#	F#	None	`cmpxchg8b` `crc32`, `div`	`nop ebx`†	27
Angr 9.2.191	Python	Python	`bt (ZF)` , `btc (ZF)` `btr (ZF)` , `bts (ZF)` `mul (CF, OF)`	`rcr` `rcl`	`nop ebx`	19
Ghidra 12.0.4[1]	SLEIGH	Rust	(`AF`†)	`movbe`, `xadd`	None	11
Triton 1.0.0rc4	C++	Python	`popcnt` (all)	`push`, `cmpxchg`	None	21

[1] This is the version from which we forked.
† Controversial interpretation (cf. Appendix C).

Miasm[14] is a reverse engineering framework written in Python. Miasm provides binary file manipulations, disassembly, lifting, emulation, and deobfuscation.

Triton[34] is a binary analysis platform written in C++. It notably supports taint analysis, emulation, and symbolic execution. Triton runs a significant number of tests that compare it to Unicorn at every instruction while emulating a program.

Unicorn[29] is a CPU emulator framework written in C. It's based on QEMU and supports multiple architectures, including x64.

4.2 Case study

Table 1 contains a summary of all 35 semantic errors we found. For each tool, along with its version, we mention the language used by the developers to describe the semantics[8] and the language we used to write the tests. We then list the errors, categorized by whether they affect instruction disassembly, flags, or registers. We also report the number of warnings raised by BiNit which we treated as false positives, as explained in Section 5.

We investigated some of these errors and opened 13 issues. Each issue can encompass multiple failures in our tests. For similar instructions, we often found they shared errors. Example: the `BT`, `BTC`, `BTS`, `BTR` instructions all shared a `ZF` bug in `angr`.

Each error was found using emulation but manually confirmed to be present *in the translated IR*. We provide details, as well as the link to the relevant issue, in Appendix C. Below is an overview of each error type, with a single example.

Register error in B2R2: we found an error with the `crc32 eax, bl` instruction. The destination value was incorrect because of an uninitialized variable in the IR.

Ordering error in Triton: we found an error for the `cmpxchg` instruction if the destination register was `AL`. This instruction reads and writes `RAX` multiple times. There was an ordering issue in the write order.

Decoding error in Unicorn: we found an error with the `ret` instruction. The immediate value was interpreted as signed rather than unsigned.

Flag error : this is the most anticipated bug (and arguably the least interesting). For example, in `angr`, the MUL instruction did not correctly calculate the `OF` and `CF` flags.

Disassembly error : although not designed to test disassemblers, we did find some disassembly issues. Notably, in Binsec, the `popcnt ax,bx` instruction could not be disassembled.

5 False positives

Although BiNit can identify many differences between the physical and emulated CPU states, these differences do not always indicate lifting errors. These are listed in Table 1 in the "Ignored Warnings" column, meaning warnings raised by BiNit that we decided not to report. We exclude these warnings for two main reasons.

First, if the instruction falls outside the lifter contract. Some lifters will raise an error if the instruction isn't implemented; for example, in Triton, `swapgs` will trigger `FAULT_UD`. This is not seen as an error, as there is direct user feedback.

Other differences can be explained by undefined behavior. Although most undefined behavior is easy to identify, sometimes output values are undefined only for specific inputs, making filtering them significantly more challenging. For example, with the `shrd` instruction: *"If the count is greater than the operand size, the result is undefined."*

For these reasons, we need to verify each failure case manually. As BiNit is intended for developers to test their own tool, we believe the overhead is reasonable. Indeed, we presume that authors will

[8] The most frequent case is a structure embedded in the program itself. In that case, we mention the programming language itself. But Ghidra, for example, has a dedicated SLEIGH language.

be familiar with their code base and will test incrementally during development, making this process significantly easier.

6 Limitations

Our memory handling is very limited; it does not allow us to test instructions that operate in multiple memory locations (such as string operations). However, it could easily be improved by adding additional phony registers.

Similarly, our CFG tests are limited. Instead of allowing jump addresses, we should deny mapped pages. Indeed, as this is a custom kernel, we have full control over which pages are mapped and which aren't.

7 Related work

The importance of lifter accuracy and lifter testing has already been widely discussed in Kim *et.al.*'s paper Testing Intermediate Representations for Binary Analysis. They introduce *N-Version IR testing*, where multiple IRs are translated to the same *Universal IR*, and then their semantics are compared. This solution is very elegant and allows them to abstract away from ground-truth issues. However, it suffers from a few drawbacks. As mentioned in their paper, it may miss bugs if both binary lifters under test have a faulty implementation. Another major challenge is the translation itself. Each IR needs to be translated to UIR before comparisons, which requires writing a complex, potentially buggy translator from IRs to UIR. Out of the three IRs they tested, it took 300 lines of OCaml code for BAP and 2,000 lines for `angr`.

Regarding lifter testing, work is underway to verify binary-analysis lifters formally. BIRD [18] uses a formally described IR to prove that mutations of the IR are correct. However, they still assume that the translation from x86 to the first version of their IR is correct.

Alternatively, work [36, 43] has been done to evaluate the correctness of binary analyzers as a whole. For example, D-Helix [43] tests decompilers by comparing symbolic models of the IR from the original binary with one obtained after decompilation and recompilation. As such, they focus more on finding bugs in IR transformation passes, whereas we focus directly on translating from assembly to IR. Alternatively, Cornucopia [36] allows for testing of binary analyzers by automatically generating a dataset of multiple binaries compiled from the same source code with different compiler options. Such efforts are complementary to our approach and can be combined to increase confidence in the correctness of a binary analyzer.

Finally, our approach is heavily inspired by work on CPU testing [13, 16, 42]. Tools such as Sandsifter[16] search for hidden instructions and hardware bugs in processors. Notably, LibLisa[13] automatically scans the instruction space, discovers instructions, and synthesizes their semantics. However, a single run can take more than three months. Indeed, Liblisa makes no assumptions about the underlying architecture, so it must discover each instruction and its semantics.

8 Future work

We plan on adding additional memory testing operations. Notably, string operations and instructions with the `REP` prefix. For that, the simplest solution would be to add additional phony memory "registers". However, if we want to test large memory writes, we might have more success by hashing memory pages.

Another important addition would be testing other architectures, such as ARM and RISC-V. This would require writing a kernel in those assembly languages. Indeed, a lot of our kernel code is Intel-specific and, as such, not very reusable. Comparatively, the work required to add test shapes for another architecture is negligible.

9 Conclusion

In this paper, we presented BiNit our framework for testing x64 lifters. BiNit works by comparing a single instruction run on a physical CPU to an emulated one after lifting. To facilitate integration, these tests are available in an SQL database. Using BiNit, we identified 35 bugs across 7 popular binary analysis frameworks. This shows that lifting errors remain common even in mature binary analysis frameworks.

Acknowledgments

This work was supported through CIFRE agreement 2025/0101, managed by the ANRT, in collaboration with Thales. We thank Benoît Amiaux for his support throughout the project, as well as Kyle Elliott and Eléonore Goblé for their insightful discussions during the early stages of development. We want to thank the reviewers for their helpful comments, which significantly improved the paper.

References

[1] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86—A platform for analyzing x86 executables. In *Proceedings of the 14th International Conference on Compiler Construction* (Edinburgh, UK) *(CC'05)*. Springer-Verlag, Berlin, Heidelberg, 250–254. doi:10.1007/978-3-540-31985-6_19

[2] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An open toolbox for adaptive WCET analysis. In *IFIP International Workshop on Software Technologies for Embedded and Ubiquitous Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 35–46.

[3] Sébastien Bardin and Philippe Herrmann. 2011. OSMOSE: automatic structural testing of executables. *Softw. Test. Verif. Reliab.* 21, 1 (March 2011), 29–54. doi:10.1002/stvr.423

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41–46.

[5] Vector 35 2016. *Binary Ninja*. Vector 35. https://binary.ninja/

[6] Philippe Biondi, Raphaël Rigo, Sarah Zennou, and Xavier Mehrenberger. 2017. BinCAT: purrfecting binary static analysis. In *Symposium sur la sécurité des technologies de l'information et des communications*. Association STIC, Rennes, France, 97–118. https://github.com/airbus-seclab/bincat

[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: a binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 463–469. https://github.com/binaryanalysisplatform/bap

[8] Carnegie Mellon University, Software Engineering Institute. 2016. Pharos. Binary analysis tools. Accessed: 2026-05-05. https://github.com/cmu-sei/pharos

[9] Silvio Cesare and Yang Xiang. 2012. Wire – A Formal Intermediate Language for Binary Analysis . In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE Computer Society, Los Alamitos, CA, USA, 515–524. doi:10.1109/TrustCom.2012.301

[10] Michael Chesser, Surya Nepal, and Damith C. Ranasinghe. 2023. Icicle: A Redesigned Emulator for Grey-Box Firmware Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 76–88. doi:10.1145/3597926.3598039

[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* 46, 3 (2011), 265–278. https://github.com/avatarone/s2e

[12] Wonkeun Choi. 2025. *Airlift: A Binary Lifter Based on a Machine-Readable Architecture Specification*. Master's thesis. Technische Universität München.

[13] Jos Craaijo, Freek Verbeek, and Binoy Ravindran. 2024. libLISA: Instruction Discovery and Analysis on x86-64. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 283 (Oct. 2024), 29 pages. doi:10.1145/3689723

[14] Fabrice Desclaux. 2012. Miasm: Framework de reverse engineering. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*. Association STIC, Rennes, France, 368–392. https://github.com/cea-sec/miasm

[15] Adel Djoudi and Sébastien Bardin. 2015. BINSEC: Binary Code Analysis with Low-Level Regions. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. Springer-Verlag, Berlin, Heidelberg, 212–217. doi:10.1007/978-3-662-46681-0_17

[16] Christopher Domas. 2017. Breaking the x86 ISA. In *Black Hat*.

[17] Thomas Dullien and Sebastian Porst. 2009. *REIL: A platform-independent intermediate representation of disassembled code for static code analysis*. Zynamics Gmbh. https://www.zynamics.com/downloads/csw09.pdf

[18] Daniel Engel, Freek Verbeek, and Binoy Ravindran. 2023. BIRD: A Binary Intermediate Representation for Formally Verified Decompilation of X86-64 Binaries. In *Tests and Proofs*, Virgile Prevosto and Cristina Seceleanu (Eds.). Springer Nature Switzerland, Cham, 3–20.

[19] Emmanuel Fleury, Olivier Ly, Gérald Point, and Aymeric Vincent. 2015. Insight: An Open Binary Analysis Framework. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. Springer-Verlag, Berlin, Heidelberg, 218–224. doi:10.1007/978-3-662-46681-0_18

[20] Kathryn E Gray, Peter Sewell, Christopher Pulte, Shaked Flur, and Robert Norton-Wright. 2017. *The Sail instruction-set semantics specification language*. Technical Report. Technical report published by Cambridge University.

[21] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified synthesis: automatically learning the x86-64 instruction set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 237–250. doi:10.1145/2908080.2908121

[22] hex-rays 1991. *Ida*. hex-rays. https://hex-rays.com/ida-pro

[23] Intel 2025. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Intel.

[24] Intel XED developpers. 2026. *Intel XED*. Intel. https://github.com/intelxed/xed

[25] Minkyu Jung, Soomin Kim, HyungSeok Han, Jaeseung Choi, and Sang Kil Cha. 2019. B2R2: Building an Efficient Front-End for Binary Analysis. In *Proceedings of the NDSS Workshop on Binary Analysis Research (BAR)*. Internet Society, Fredericksburg, VA. 22401, U.S.A, 10 pages. doi:10.14722/bar.2019.23051

[26] Johannes Kinder and Helut Veith. 2008. Jakstab: A Static Analysis Platform for Binaries. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 423–427. https://github.com/jkinder/jakstab

[27] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2020. Manticore: a user-friendly symbolic execution framework for binaries and smart contracts. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, Washington, D.C., United States of America, 1186–1189. doi:10.1109/ASE.2019.00133

[28] National Security Agency. 2019. Ghidra. Reverse engineering framework. Accessed: 2026-05-05. https://github.com/NationalSecurityAgency/ghidra

[29] NGUYEN Anh Quynh and DANG Hoang Vu. 2015. *Unicorn: Next generation CPU emulator framework*. https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf

[30] radare2 Project. 2006. radare2. Reverse engineering framework. Accessed: 2026-05-05. https://github.com/radareorg/radare2

[31] radare2 Project. 2026. radare2. Documentation for radare2. Accessed: 2026-05-05. https://book.rada.re/

[32] Matthew Revelle, Matt Parker, and Kevin Orr. 2023. Blaze: A Framework for Interprocedural Binary Analysis. In *Proceedings 2023 of the NDSS Workshop on Binary Analysis Research*. Internet Society, Fredericksburg, VA. 22401, U.S.A, 11 pages. https://github.com/kudu-dynamics/blaze-platform

[33] Rizin Project. 2020. Rizin. Fork of radare2. Accessed: 2026-05-05. https://github.com/rizinorg/rizin

[34] Florent Saudel and Jonathan Salwan. 2015. Triton: Concolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*. Association STIC, Rennes, France, 31–54. https://github.com/jonathansalwan/Triton

[35] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, Washington, D.C., United States of America, 138–157. doi:10.1109/SP.2016.17

[36] Vidush Singhal, Akul Abhilash Pillai, Charitha Saumya, Milind Kulkarni, and Aravind Machiry. 2023. Cornucopia: A Framework for Feedback Guided Generation of Binaries. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 27, 13 pages. doi:10.1145/3551349.3561152

[37] M. A. Solovev, M. G. Bakulin, M. S. Gorbachev, D. V. Manushin, V. A. Padaryan, and S. S. Panasenko. 2019. Next-Generation Intermediate Representations for Binary Code Analysis. *Program. Comput. Softw.* 45, 7 (Dec. 2019), 424–437. doi:10.1134/S0361768819070107

[38] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International conference on information systems security*, R. Sekar and Arun K. Pujari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–25.

[39] sonicwall. 2017. new anti-vm technique observed in the wild. https://www.sonicwall.com/pt-br/blog/new-anti-vm-technique-observed-in-the-wild-oct-05-2017

[40] Trail of Bits. 2017. Remill. LLVM-based binary lifting framework. Accessed: 2026-05-05. https://github.com/lifting-bits/remill

[41] Julien Vanegue. 2008. *Static binary analysis with a domain specific language*. https://www.youtube.com/watch?v=7T5oM4KyMuI

[42] ZehMatt. 2024. *x86Tester*. BinPrey. https://github.com/ZehMatt/x86Tester

[43] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian. 2024. D-Helix: a generic decompiler testing framework using symbolic differentiation. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 23, 18 pages.

A Artifact

- The database with instruction inputs/outputs is available online at https://binit.wazab.in/about;
- The code for the kernel is available at https://github.com/thalium/aegis;
- The testing harness is available at https://github.com/thalium/aegis-binit.

The versions used for this paper are tagged as SURE.

B Lifters comparison.

Table 2 lists many Intel x64 lifters, along with their IR, use case, date of release, current status, and implementation language.

C Identified errors

Resolution status. At the time of this writing, 16 bugs have been acknowledged and resolved upstream.

Controversial. The following bugs have been reported upstream, but were not acknowledged by authors due to their controversial interpretation. The `(AF)` entries for Binsec and Ghidra in table 1 are controversial because AF is architecturally undefined for the relevant instructions in 64-bit mode but well defined in 32-bit mode. The `nop ebx` encoding is a reserved-NOP encoding; B2R2 intentionally rejects it as invalid, rather than treating it as a semantic error.

Details. The numbers in parentheses are links to the pull requests of the different bug reports.

- **angr—RCL/RCR:** rotation counts were not masked. The fix masks the count before computing the rotation (#6067).
- **angr—MUL CF/OF:** the unsigned-multiply helper derived CF and OF from the low product's sign bit. The fix widens operands before extracting the high half (#6067).
- **angr—BT/BTC/BTR/BTS:** lifting replaced the lazy flags state with CF alone, spuriously clearing ZF. The correction materializes the old flags and replaces only CF (#6067).
- **B2R2—CMPXCHG8B:** a conditional-expression encoding turned the successful-path "no write" to EAX/EDX into a zero-extending write. The fix uses explicit control flow for the conditional update (#337).
- **B2R2—CRC32:** the `bitReflect` helper used an uninitialized IR temporary. Initializing it to zero fixes the result (#338).
- **B2R2—DIV:** `0xffffffff` was inferred as an F# `Int32` and sign-extended in the 64-bit path. Using a 64-bit literal fixes the division semantics (#340).
- **Binsec—POPCNT:** the model did not clear the flags required by the instruction, including CF. The issue was assigned to `unisim_archisec` (#61).
- **Binsec—IMUL:** the immediate was extended only to 64 bits before a 128-bit multiplication, yielding incorrect overflow flags for negative immediates. It should be sign-extended to the computation width (#61).
- **Binsec—SAR and SHRD flags:** flags were computed from the pre-shift destination rather than the final result. They should be computed after the shift/merge operation (#61).
- **Binsec—decoding:** `popcnt ax,bx` was unsupported, `movzx rax,bx` used a 32-bit source, and the `F2` CRC32 encodings were decoded as MOVBE. These require width- and prefix-aware decoder entries (#61).
- **Triton—POPCNT flags:** POPCNT left flags unchanged instead of clearing CF, OF, SF, AF, and PF and setting ZF from the result (#1444).
- **Triton—CMPXCHG with AL:** multiple reads/writes of RAX were emitted in the wrong order, so the AL destination was not updated. The update must use the pre-instruction accumulator value (#1448).
- **Triton—PUSH with an operand-size prefix:** immediate PUSH used the stack-pointer width rather than the encoded operand width. The stack adjustment must use the operand size (#1450).
- **Unicorn—BLSI:** CF had inverted/incorrect source-zero semantics. The x86 BLSI carry-flag helper was corrected (#2330).
- **Unicorn—BZHI:** the index-boundary case was handled incorrectly when the index was at least the operand width. The boundary semantics were corrected (#2331).
- **Unicorn—CMPXCHG:** the accumulator was updated even on the successful comparison path. The accumulator-update semantics were corrected (#2332).
- **Unicorn—PDEP:** 32-bit PDEP used an incorrect mask width and failed to clear the upper half of RAX. The 32-bit mask handling was fixed (#2334).
- **Unicorn—RORX, SHLD, and SHRD:** RIP-relative addressing with a trailing immediate used an address one byte too early. The effective-address computation was corrected (#2335).
- **Unicorn—RET `imm16`:** the unsigned stack-adjustment immediate was sign-extended, rather than zero-extended. The `imm16` adjustment was corrected (#2336).
- **Miasm—ADCX/ADOX decoding:** opcode matching selected WRSS encodings instead of ADCX/ADOX. The decoder needs disambiguated, prefix-aware instruction matching (#1528).
- **Miasm—POPCNT disassembly:** Miasm 0.1.5 does not decode `POPCNT`. No upstream issue was opened for this finding.
- **Ghidra—MOVBE:** the 32-bit load form `MOVBE Reg32,m32` wrote only the `Reg32` varnode, leaving the upper half of the underlying 64-bit register unchanged. In 64-bit mode, a 32-bit destination write must zero-extend. The fix adds `check_Reg32_dest`, which emits the required full-register zero-extension (Open SLEIGH commit).
- **Ghidra—XADD:** the lockable 32-bit memory form `XADD m32,Reg32` updates `Reg32` with the old memory value, but its constructor omitted the 32-bit-destination zero-extension check. Thus, the upper 32 bits of the corresponding 64-bit register could remain stale. The fix adds and builds `check_Reg32_dest` after the `xadd` p-code operation (Open SLEIGH commit).

D Generative AI Usage

This paper was edited for grammar using Grammarly.

LLMs (Claude and Codex) were used for code generation in this project. The frontend of the DB's website was entirely generated, as were the SQL-to-lifter adapters, as mentioned in the paper. We manually replicated every bug and then handwrote the issues in GitHub. A real human was behind each interaction with a project author. The detailed summaries of pull requests were generated by IA and manually validated by the authors.

Table 2: Comparison of intermediate representations used in binary analysis frameworks.

IR	Framework	Reference	Use case	Release	Status	Language
PCode	Ghidra	[28]	decompilation	2019	Active	C++/Java
esil	radare2	[30]	human-readable, partial emulation	2006	Active	C
VEX	angr	[35]	vulnerability discovery, deobfuscation, decompilation	2016	Active	Python
RzIL	rizin	[33]	human-readable, partial emulation	–	Active	C
LLVM 15+	remill	[40]	accurate lifting to LLVM	2017	Active	C++
–	Pharos	[8]	–	2015	Active	C++
LowUIR	B2R2	[25]	analysis	2019	Active	F#
DBA	BINSEC	[15]	vulnerability discovery, analysis, deobfuscation	2015	Active	OCaml
Triton AST	Triton	[34]	deobfuscation	2015	Active	C++
–	Miasm	[14]	emulation, analysis, deobfuscation	2012	Active	Python
–	S2E	[11]	path exploration	2011	Active	C
PIL	Blaze	[32]	vulnerability discovery	2023	Active	Haskell
BIL	BAP	[7]	analysis	2011	Inactive	OCaml
–	BinCAT	[6]	analysis	2017	Inactive	OCaml
Jakstab	Jakstab	[26]	analysis, recover CFG	2008	Discontinued	Java
REIL	OpenREIL	[17]	analysis	2009	Discontinued	–
GAL	OSMOSE	[3]	–	2008	Discontinued	–
–	Manticore	[27]	–	2019	Discontinued	Python
–	OTAWA	[2]	–	2010	Discontinued	–
microcode	Insight	[19]	–	2012	Discontinued	–
AIR	Airlift	[12]	correctness, verification	2025	Unknown	Rust
–	BINCOA	–	–	–	Unknown	–
Vine	BitBlaze	[38]	–	2008	Unknown	C++/OCaml
–	CodeSurfer/x86	[1]	–	2005	Unknown	–
bird	BIRD	[18]	formally verified decompilation	2023	Unknown	–
wire	Wire	[9]	–	2012	Unknown	–
pivot2	Pivot2	[37]	–	2019	Unknown	–
ERESI	–	[41]	–	2008	Discontinue	–