B2R2: Building an Efficient Front-End for Binary Analysis

Minkyu Jung*, Soomin Kim*, HyungSeok Han*, Jaeseung Choi*, Sang Kil Cha*

*KAIST
{hestati, soomink, hyungseok.han, jschoi17, sangkilc}@kaist.ac.kr

***Abstract*—Current binary analysis research focuses mainly on the back-end, but not on the front-end. However, we note that there are several key design points in the front-end that can greatly improve the efficiency of binary analyses. To demonstrate our idea, we design and implement B2R2, a new binary analysis platform that is fast with regard to lifting binary code and evaluating the corresponding IR. Our platform is written purely in F#, a functional programming language, without any external dependencies. Thus, it naturally supports pure parallelism. B2R2's IR embeds metadata in its language for speeding up data-flow analyses, and it is designed to be efficient for evaluation. Therefore, any binary analysis technique can benefit from our IR design. We discuss our design decisions to build an efficient binary analysis front-end, and summarize lessons learned. We also make our source code public on GitHub.**

I. Introduction

Binary analysis is crucial in software security, and numerous tools for it are available nowadays. For example, GitHub currently accommodates hundreds of public repositories[1], the descriptions of which contain the term "binary analysis".

Binary analysis tools typically consist of two major components: the front-end and the back-end. The front-end, which mainly consists of a disassembler and a lifter, disassembles a given binary and translates it into what is known as an *Intermediate Representation* (IR). The back-end takes in the lifted IR as input, and performs the actual analysis on it, with examples being CFG recovery [30], structural analysis [39], [48], and type inference [31], [38]. All the existing frameworks, whether they are open-source [4], [13], [17], [18], [41] or closed-source [7], [27], [47], employ their own IR. For example, IDA Pro [27] uses Microcode [25], and angr [40] operates with VEX IR [34].

As the back-end depends on the front-end as well as its IR, the performance of the front-end can obviously affect the efficiency of a binary analysis. Static binary analyses rely on Control-Flow Graph (CFG) recovery techniques [17], [30], which essentially require lifted IRs to start with. Dynamic symbolic executors [22], [44] and dynamic taint trackers [35] can directly benefit from employing an efficient binary lifter as they perform lifting for every instruction encountered during program execution. Yun *et al.* [49] recently confirmed that the performance of lifting can be a bottleneck of symbolic executors.

However, researchers have paid a little attention to the designs of front-ends since binary lifting can be considered as a simple translation process. Several efforts to design a concise and easy-to-use IR have been made [10], [13], but not in relation to designing an IR that can be quickly evaluated. Many researchers have rather focused on devising effective back-end algorithms [17], [24], [30], [31], [38], [39].

In this paper, we show that there are numerous points in the design of the front-end and the corresponding IR that can greatly benefit both the front-end itself and the back-end in terms of their efficiency.

To name a few, we found that translating a binary into a well-optimized IR is difficult, and employing a simple local optimizer can greatly reduce the complexity of IRs. However, such optimization passes can slow down the entire lifting process as they must iteratively traverse the corresponding Abstract Syntax Trees (ASTs). For example, the optimization pass in our system occupies more than one third of the total running time of the front-end. We can mitigate this problem by exploiting multi-core parallelism, but it is not straightforward to disassemble instructions in parallel as their size varies depending on the opcode. Thus, we propose a novel technique for lifting binary instructions in parallel.

We also found that the structure and implementation of an IR can drastically affect the performance of IR evaluations and binary analyses. For example, existing IRs represent a number with arbitrary-precision integers because machine instructions often involve arithmetic operations between registers holding numbers larger than 64 bits, e.g., the `XMM` and `YMM` registers of x86. However, arbitrary-precision computation can substantially slow down the IR evaluation process. Additionally, one may design an IR, each expression of which incorporates useful metadata about used variables in the expression. We show that such a simple twist in the design of IR enables an efficient data-flow analysis.

To the best of our knowledge, this paper presents the first rigorous study of the design of an efficient binary analysis front-end and its IR. In particular, we design and implement a new binary analysis framework that we call B2R2, and summarize critical design decisions we made to build an efficient binary analysis front-end.

[1]https://github.com/search?q=%22binary+analysis%22&type=Repositories.

Workshop on Binary Analysis Research (BAR) 2019
24 February 2019, San Diego, CA, USA
ISBN 1-891562-58-4
https://dx.doi.org/10.14722/bar.2019.23xxx
www.ndss-symposium.org

The key contributions of this work are as follows.

1) We propose a novel IR lifting technique that exploits multi-core parallelism.
2) We study the current state-of-the-art binary analysis front-ends, and discuss their design decisions.
3) We present a novel design of an IR that can speed up the IR evaluation process.
4) We design and implement B2R2, an efficient binary analysis framework that implements our techniques and design choices.
5) We make our tool public on GitHub.

The rest of the paper is organized as follows. We first summarize several related works on designing an IR. We then present key observations on the structure of existing binary analysis front-ends. Next, we present our design choices for B2R2, our new binary analysis platform. Finally, we conclude the paper by describing our experimental results.

II. Related Work

There are a few IRs that provide their formal semantics. GAL [9] and DBA [10], for instance, focus on the ease of binary analysis by providing useful operators such as bit-manipulation operators. BAP [13] presents a concise IR that can explicitly represents all assembly side-effects. However, none of them focuses on the efficiency of IR evaluation.

Kim *et al.* [29] show the first systematized study on existing lifters. They characterize each lifters and their IRs based on their expressiveness. In particular, they formally define the two properties: explicitness and self-containment. An IR is *explicit* if every IR statement updates only a single variable at a time, and an IR is *self-contained* if it can exclusively describe the semantics of machine instructions. However, their focus was on testing the correctness of IRs, but not on designing an efficient lifter nor evaluator.

Godefroid and Taly [23] use template-based program synthesis to automatically translate x86 instructions into IRs. They utilize a CPU to collect input-output samples for each instruction, and employ a program synthesis technique to generate the corresponding IRs. Synthesis-based approach is further investigated in [46], where the authors leveraged existing IR translations to mine templates for synthesis. In this work, they used a program synthesis technique to extend a lifter to support previously unsupported architectures. Hasabnis and Sekar [26] adopt a learning-based approach to automate lifting. Their approach leverages a compiler to generate a dataset for IR-to-assembly translation, and use the dataset to learn a mapping from the assembly to the IR. While these works aim to reduce the human efforts in implementing lifters, we rather focus on designing a highly optimized lifter.

III. Preliminary Study

Our research is inspired by examining existing binary analysis tools. In particular, we picked 10 open-source tools that do not rely on COTS software, and analyzed their front-end. Table I summarizes the characteristics of their lifter and their IRs. Note we omit McSema [45] here because it is dependent on IDA Pro [27].

A. Observation

By reviewing the tools, we found several crucial points in the design of binary analysis front-ends. We highlight them here in the order of columns appeared in Table I. All these observations boil down to the design of B2R2, which are described in §IV.

O1 (**Parallelism**). Many binary analysis tools do not support pure parallelism due to their language choice. The half of the tools use either Python (CPython) or OCaml, which cannot take advantage of multiple cores [11]. Although the impact of parallelizability of the front-end has been neglected, we show that pure parallelism can greatly improve the performance of it: our parallel lifting technique increases the overall lifting speed up to twice on a modern desktop computer (see §V-B).

O2 (**IR Optimization**). Only few tools such as BAP [13] and PyVEX [3] perform IR optimization while lifting binaries. IR optimization does not help in reducing analysis costs unless there is a greater gain from evaluating the lifted IR statements. However, we show that one can achieve significant performance improvement for IR optimization by exploiting multi-core parallelism: the lifting throughput can be better than the one without IR optimization (§V-B).

O3 (**AST Construction**). Most tools use Abstract Syntax Trees (ASTs) to represent their IRs and to perform back-end analyses. However, there is one notable exception: radare2 [4] does *not* build ASTs during the lifting process. Instead, it simply emits a string representation for their IR. Such a design choice enables efficient lifting, but, on the other hand, it makes writing a static analyzer difficult.

O4 (**Metadata in AST**). Most existing IRs do not store any extra information in their ASTs. Notably, PyVEX [3] preserves type information of branch statements (PyVEX refers to it as a jump kind) in their ASTs. However, we are not aware of any existing IR that keeps metadata in their ASTs to help binary analysis. In this paper, we present a novel AST construction methodology that enables efficient data-flow analysis by embedding metadata into IR expressions (see §IV-D).

O5 (**IR Expressiveness**). One can represent the expressiveness of IRs with the concept of self-containment and explicitness [29], Most IRs are both explicit and self-contained, but some of them are not. Notably, VEX IR uses external C functions to express instruction semantics, which makes it difficult to write a static analyzer with the IR. However, the expressiveness of an IR can substantially impact the ease of writing a binary analyzer. As an example, the symbolic execution engine of angr, which internally utilizes VEX IR, contains handlers for the external functions with 1.7 KLOC.

O6 (**Hash-Consing**). We found that none of the existing binary analysis tools natively support hash-consing for their IR. Hash-consing [20] is widely known to be crucial in building a large-scale symbolic execution system [6], [12] as it allows structurally the same ASTs to be stored in the same physical memory, thereby significantly reducing the memory use. However, none of the existing tools provides hash-consing feature for their IRs.

O7 (**Arbitrary-Precision Integers**). Most existing tools employ big integers, i.e., arbitrary-precision integers, in their IR implementation. Arbitrary-precision integers help represent

TABLE I: Existing open-source tools[1] for binary analysis.

		Lifter Design				IR Characteristics						Architecture Support						
Tool	**IR Name**	**Programming Language**	**Pure Parallelism**	**IR Optimization**	**AST Construction**	**Metadata Embedding**	**Explicit[5]**	**Self-Contained[5]**	**Hash-consed IR Support**	**x86 SIMD Support**	**Big Integer Splitting**	**x86**	**x86-64**	**ARMv7**	**ARMv8**	**Thumb**	**MIPS32**	**MIPS64**
angr [40]	VEX[2]	C & Python	✗	✓	✓	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
BAP [13]	BIL	OCaml[4]	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
BINSEC [18]	DBA	OCaml	✗	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗
BinNavi [19]	REIL	Java	✓	✗	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
BitBlaze [41]	Vine	C & OCaml	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗
Insight [21]	Microcode[3]	C++	✓	✗	✓	✗	✓	✓	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗
Jakstab [30]	SSL	Java	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗
Miasm [15]	Miasm IR	C & Python	✗	✗	✓	✗	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✗
radare2 [4]	ESIL [1]	C	✓	✗	✗	✗	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
rev.ng [17]	LLVM	C++	✓	✗	✓	✗	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
B2R2[★]	LowUIR	F#	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

[1] We intentionally omit MCSema [45] as it relies on IDA Pro for disassembling binaries.
[2] angr internally uses VEX IR, and it lifts binaries using a module called PyVEX [3], which is a Python wrapper of the VEX lifter [34] originally written in C.
[3] This should not be confused with IDA Pro's IR, which is also referred to as Microcode.
[4] The ocaml-multicore project [28] aims to support multicore parallelism, but it is not yet integrated into the upstream.
[5] We follow the definition of explicitness and self-containment in [29].
[★] This is our work.

SIMD instructions, which use vector registers such as the `XMM` and `YMM` registers of Intel, holding values greater than 64 bits. We note that most binary analysis tools support SIMD instructions, but none of them split big integers into smaller chunks to avoid the use of arbitrary-precision integers. However, arbitrary-precision arithmetic is considerably slower than fixed-precision arithmetic, which can be directly run on an ALU. Our study shows that implementing the semantics of machine code with fixed-precision integers is indeed straightforward, and can improve the speed of IR evaluation (§V-C).

O8 (Architecture Support). Many tools focus on a small set of Instruction Set Architectures (ISAs) as one needs to write a lifter for each different ISA independently, which requires significant engineering effort. This observation encourages the modular development of front-ends (§IV-A) as a binary analysis framework can easily adopt another lifter in its front-end. The last column of Table I demonstrates how each tool supports several major architectures.

B. Experimental Setup

To further understand the characteristics of the binary analysis tools and their front-ends, we ran each of the tools to lift a set of real-world binaries obtained from Ubuntu 16.04. Our main aim here is to compare the lifting throughput of the tools in Table I. To solely measure the performance of lifting, we ran our test binaries with each tool in a linear-sweep fashion without performing extra IR optimization nor back-end analyses such as function boundary identification or CFG reconstruction. Although we ran lifters only in a linear-sweep manner, the measured lifting throughputs can equally affect the performance of recursive-descent lifters. We chose not to use recursive-descent lifting in our experiments as not every tool supports it.

To gather the binaries, we first set up both x86 and x86-64 Ubuntu on two fresh new VMs. We then extracted binaries located at `/bin`, and `/usr/bin` from each VM. As a result, we obtained 1,200 and 1,204 ELF binaries in total from the x86 and the x86-64 VM, respectively.

We note that some binary lifters such as PyVEX can only take in a raw binary as input, whereas other tools such as BINSEC can only take in a well-formed binary, e.g., an ELF binary, as input. Since adding a support to a tool for parsing ELF or PE binaries is time-consuming and error-prone, we decided to use only raw binaries for our experiments. Specifically, we extracted the `.text` section of each of the obtained binaries with `objcopy`, and combined all of them into a single raw binary file for 32- and 64-bit architecture, respectively. As a result, we obtained 78MB blob and 79MB blob for 32-bit and 64-bit architecture, respectively. We also obtained one more extra raw binary by extracting the `.text` section of a GNU C library (LIBC) file from the x86 VM. In total we have three blobs of raw binary instructions as indicated in Table II.

Although we originally tried to run all the tools we studied, we were only able to run five of them, because we had to modify significant amount of code to run the other tools without performing extra analyses. Table II lists the five tools that we were able to run. The third column of the table indicates the release date of each version of the tools that we used. We still had to manually modify some of the lifters in the table, and we summarize all the modifications that we made as follows.

PyVEX. We disabled IR optimization by modifying 3 lines of PyVEX. We also wrote a Python script of 23 lines that lifts a sequence of binary instructions to a sequence of VEX IRs.

BAP. We used `bap-mc`, a stand-alone command-line tool, in order to avoid running other analyses while lifting. We applied a one-line patch to `bap-mc` to ignore unhandled instructions.

BINSEC. BINSEC does not take in a raw binary file as input. Thus, we modified 38 lines of BINSEC to support lifting the raw binary files.

Miasm. We wrote a Python script of 23 lines, which takes in

TABLE II: Lifting time comparison.

Tool	Version	Latest Release	Lifting Time (sec.) LIBC	x86 Blob	x86-64 Blob
angr	8.18.10.25	2018/10/25	39.6	2576.6	2360.2
BAP	1.5.0	2018/10/11	55.3	2558.8	1746.1
BINSEC	0.2	2018/10/1	25.5	2078.7	N/A[1]
Miasm	0.1.0	2018/11/12	171.5	9581.6	8686.5
radare2	3.1.3	2018/12/5	1.8	104.1	94.3
B2R2	0.0.1	First prototype	11.4	568.8	455.9
B2R2	0.1.0	★ our work	3.5	201.4	169.5

[1] BINSEC only supports x86 architecture.

a raw binary as input, and lifts it to a sequence of Miasm IR statements.

radare2. We used `rasm2`, a stand-alone command-line tool, to get ESIL strings from the raw binary files. We patched one line of `rasm2` to ignore unhandled instructions.

C. Performance Comparison of Lifters

We ran the five tools on a single Intel 2.10 GHz Xeon E5 core with the three different raw binary blobs we obtained in §III-B. The right-most column of Table II shows the lifting time comparison between the tools. The result was rather surprising because the lifting throughputs differ by two orders of magnitude in the worst case: the fastest was radare2, and it was more than $100\times$ faster than Miasm in lifting the binaries. Despite the fact that radare2 is written in C, it was still substantially faster than the other tools. Of course, being a fast lifter does not mean that radare2 is the best binary lifter: it does not build ASTs as discussed in O3, and it is relatively difficult to write an analyzer with it.

Our preliminary study shows that most existing tools rather focus on the ease of analysis, but not on the efficiency of their front-end engines. Indeed, this is one of the key motivation that inspired our research. Can we design a binary analysis front-end on which one can easily write an analyzer, while being still efficient? It is widely believed that writing a program analyzer with a functional language that supports algebraic data types and pattern matching [43] is significantly easier than that with other languages. Therefore, our goal here is to write an efficient binary analysis front-end while following a functional paradigm.

O9 (**Lifting Performance**). Most existing tools do not focus on optimizing the lifting throughput. However, there is huge room for improvement with regard to the speed of lifting, and one can substantially boost up the speed with a proper engineering effort. As shown in Table II, we could make B2R2 $2.8\times$ faster than its first prototype on our dataset, thanks to the heavy optimizations we made on our system (§IV-H).

In this paper, we designed and implemented a new binary analysis framework that we call B2R2. To compare the efficiency of B2R2 with that of existing tools, we ran the optimized version of B2R2 on the same dataset, without enabling our parallel lifting technique. Our experimental result shows that B2R2 was only $1.9\times$ slower than radare2 while still providing the ease of analysis. When we compare the performance of B2R2 against BAP, BINSEC, and angr (PyVEX), and it was $11.6\times$, $10.3\times$, and $13.3\times$, faster, respectively.

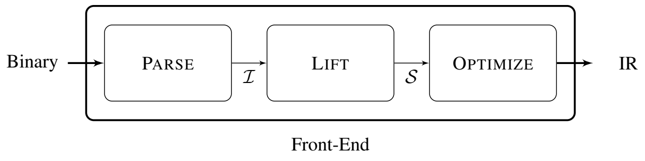


Fig. 1: Architecture of the B2R2 front-end.

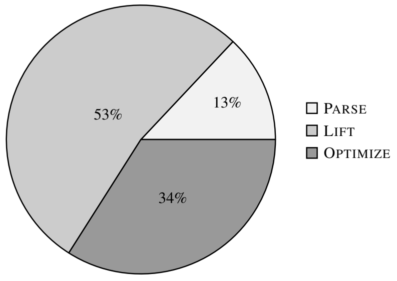


Fig. 2: Performance breakdown of B2R2's front-end.

IV. B2R2 DESIGN

In this section, we present the overall design and implementation of B2R2 and its IR. Particularly, we enumerate key design choices for building an efficient binary analysis front-end and its IR.

A. Modular Front-End Design

B2R2 splits the front-end into the three main modules: PARSE, LIFT, and OPTIMIZE. Figure 1 depicts the overall architecture of the B2R2 front-end, which takes in a binary as input and returns a lifted IR as output. PARSE first parses given binary code to obtain a data structure $\mathcal{I}$ representing a sequence of instructions in an architecture-neutral manner. LIFT then lifts the given instructions to our own IR statements $\mathcal{S}$, i.e., LowUIR, and finally, OPTIMIZE optimizes the IR statements to produce the final IR statements.

We observe the similar design pattern from other binary analysis tools such as BAP [13]. However, some tools do not follow a modular design approach, which makes it difficult to run each of the steps independently. For instance, PyVEX [3] embeds the IR optimization process in its lifting function. Our modular design approach is one of the key ingredients for enabling parallel lifting (§IV-B).

B. Parallel Lifting and Optimization

Parallel lifting is particularly challenging on Intel architectures, i.e., x86 and x86-64, as they encode instructions with a byte sequence of variable length. That is, we cannot predict the next instruction unless we completely parse the current instruction, and figure out the size of it. Therefore, the parsing step (PARSE) cannot run in parallel.

However, given a list of parsed instructions, one can easily lift them in parallel as there is no dependency between them. B2R2 first accumulates parsed instructions from PARSE, and performs lifting (LIFT) and optimization (OPTIMIZE) on them in parallel. On the other hand, existing lifters run PARSE

METADATA	$\boldsymbol{\mu}$	::=	`ExprInfo` * `ConsInfo`
		|	`ExprInfo`
ENDIAN	$\boldsymbol{\epsilon}$	::=	`BEndian` | `LEndian`
UNOP	$\Diamond_u$	::=	`NEG` | `NOT`
BINOP	$\Diamond_b$	::=	`ADD` | `SUB` | `MUL` | `DIV` | `SDIV` | `MOD` | `SMOD` | `SHL` | `SHR` | `SAR` | `AND` | `OR` | `XOR` | `CONCAT`
RELOP	$\Diamond_r$	::=	`EQ` | `NEQ` | `GT` | `GE` | `SGT` | `SGE` | `LT` | `LE` | `SLT` | `SLE`
CASTOP	$\Diamond_c$	::=	`Extract` | `ZeroExt` | `SignExt`
EXPRESSION	$\boldsymbol{exp}$	::=	`Num` *value size*
		|	`Var` *name size*
		|	`PCVar` *name size*
		|	`TempVar` *name size*
		|	`Name` *name*
		|	`UnOp` $\Diamond_u$ $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	`BinOp` $\Diamond_b$ $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	`RelOp` $\Diamond_r$ $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	`Load` $\boldsymbol{\epsilon}$ *size* $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	`ITE` $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	`Cast` $\Diamond_c$ *size* $\boldsymbol{exp}$ $\boldsymbol{\mu}$
		|	`Undefined` *size*
STATEMENT	$\boldsymbol{stmt}$	::=	`ISMark` *addr len*
		|	`IEMark` *addr*
		|	`LMark` *name*
		|	`Put` $\boldsymbol{exp}$ $\boldsymbol{exp}$
		|	`Store` $\boldsymbol{\epsilon}$ $\boldsymbol{exp}$ $\boldsymbol{exp}$
		|	`Jmp` $\boldsymbol{exp}$
		|	`CJmp` $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{exp}$
		|	`InterJmp` $\boldsymbol{exp}$ $\boldsymbol{exp}$
		|	`InterCJmp` $\boldsymbol{exp}$ $\boldsymbol{exp}$ $\boldsymbol{exp}$
		|	`SideEffect` *SideEffect*

Fig. 3: The syntax of LowUIR.

and LIFT iteratively for each instruction, which fundamentally precludes parallel computation.

Furthermore, in our preliminary experiments (§III-C), we found that the PARSE module occupies only 13% of the CPU cycle during the entire run of the front-end. Figure 2 describes the performance breakdown for each module of B2R2's front-end. This means that 87% of the cost of the front-end can be parallelized with our approach.

In our parallel lifting implementation, we accumulate N parsed basic blocks from PARSE, and asynchronously lift and optimize the accumulated blocks of instructions on multiple threads. The number of accumulated basic blocks N is a user-configurable parameter, and an optimal value of it can differ depending on the machine. Note that we cannot simply increase the value of N as we will consume more memory, and give more pressure to our garbage collector. Automatically configuring an optimal parameter value N is beyond the scope of this paper.

We are not aware of any existing binary analysis tool that performs parallel lifting nor optimization. Recall that many existing tools are written in a language that does not support pure parallelism (O1). Therefore, it is difficult to exploit multiple cores within a single process. However, since B2R2 is written in a functional language that support immutability, we can easily achieve parallelism. In our experiments, parallel lifting can improve the performance of the entire pipeline of our front-end by twice (see §V-B).

C. LowUIR

As discussed in O5, an IR should be both explicit and self-contained in order to ease the back-end analyses. B2R2's IR, which we refer to as LowUIR, also follows the same design choice as most other IRs do. Figure 3 shows the formal description of LowUIR.

Since a single machine instruction typically corresponds to multiple IR statements, LowUIR explicitly marks the start and the end of a machine instruction with the `ISMark` and `IEMark` expression, respectively. In a similar manner, LowUIR distinguishes branch instructions based on whether the jump target is within the same instruction or not. For example, the `BSF` instruction of x86 scans each bit in a register within a loop. We can concisely write the semantics of it with an intra-instruction jump, which we denote as `Jmp` or `CJmp`. For those regular branch instructions between actual machine instructions, e.g., the `jne` instruction of x86, we use the `InterJmp` and `InterCJmp` statement. These IR statements help us explicitly identify whether a control transfer is within an instruction or not.

D. IR Metadata Embedding

Notably, each expression in LowUIR contains metadata (`ExprInfo`) about the corresponding expression. More formally, IR metadata is extra information stored in an AST that does *not* affect the operational semantics of the IR. In LowUIR, each `ExprInfo` stores (1) a set of used variables in the expression, and (2) a boolean value indicating whether the expression accesses memory. Such meta information is useful in writing a data-flow analyzer as we do not need to traverse every node in each AST when we construct a use-def chain [5]. That is, constructing a use-def chain takes *constant* time with our metadata. For example, we can write a taint analysis tool with a simple over-tainting policy without traversing the entire ASTs because we can figure out taint sources (uses) by observing the metadata, and we can quickly recognize taint sinks (defs) by checking the root nodes of the ASTs because our IR is explicit, meaning that each statement can update only a single variable.

Unfortunately, constructing such metadata can be pure overhead for lifting. In our experiments, we observed about 3% of performance degradation by adding metadata to LowUIR. On the other hand, we leverage the metadata to boost up our local optimizer (see §IV-E).

E. Block-Level Local Optimization

Writing semantics for machine instructions is fundamentally difficult, and requires significant engineering effort. Furthermore, translating machine instructions to optimized IR statements is even harder. We often observe our IR statements are not necessarily optimal: they contain redundant variable assignments or even dead code. Recall from O2, only few lifters such as PyVEX and BAP perform optimization for each basic-block they lift, which includes traditional local optimization techniques such as constant folding, dead code elimination, and various peephole optimizations [5].

B2R2 implements the three block-level local optimizations: (1) constant folding with simple algebraic simplification, (2) constant propagation, and (3) dead code elimination. We note that we can leverage meta data of our IR expressions (§IV-D) in order to efficiently perform the optimization methods. Our block-level optimization reduces the number IR statements by 17% on our dataset (see §V-A).

F. Hash-Consed ASTs

Recall from O6, none of the existing lifters natively supports hash-consed ASTs although hash-consing can greatly improve the performance of analyzers [6]. To use hash-consing on an IR that does not natively support it, one needs to wrap each AST with a new type, which potentially degrades the performance, and requires substantial engineering effort.

B2R2 implements hash-consing within the LowUIR expressions by storing a hash key as well as a unique identifier for each hash-consed AST in `ConsInfo`, which is tagged per each expression along with the instruction-level meta data `ExprInfo`. We implemented a thread-safe weak hash table [20] to maintain the hash-consed ASTs that support parallelism.

G. Bigint Splitting

Modern CPUs employ vector registers to support instruction-level parallelism. For example, Intel's Streaming SIMD Extensions (SSE) adds a set of SIMD instructions along with XMM registers such as `XMM0` that can hold 128-bit values. Although vector registers can store values of size greater than 64 bits, the ALU of x86-64 processor still operates with data of the native word size, i.e., 64 bits. Therefore, most SIMD instructions divide values in a vector register and perform multiple arithmetic operations in parallel with values of size less than 64 bits. For example, consider "`VADDPS XMM0, XMM1`" instruction of x86. This instruction splits each value stored in XMM0 and XMM1 into four 32-bit values, and adds each pair in parallel.

As we observed from O7, all the existing binary analysis tools rely on arbitrary-precision integers to represent vector registers. It is not surprising because it is straightforward to implement the semantics of SIMD instructions with arbitrary-precision integers. However, since arbitrary-precision integer arithmetics incur significant runtime overhead as they cannot run directly on an ALU.

LowUIR implements the semantics of machine instructions with only fixed-precision integers. To handle vector registers, we split them into 64-bit pseudo-registers. For example, we divide the `XMM0` register of x86 into two pseudo-registers: `XMM0A`, and `XMM0B`. We call this method of implementing the binary semantics as *bigint splitting*. Our approach is inspired by the fact that most SIMD instructions operate with small data values anyways. The downside here is that bigint splitting can produce a more number of IR statements than the one without bigint splitting. However, our empirical results show that bigint splitting helps in improving the IR evaluation speed (§V-C). Furthermore, bigint splitting can simplify the overall semantics of SIMD instructions as we do not need to explicitly extract sub-values from a vector register.

H. Implementation

B2R2 consists of 45 KLOC of F# code. It does not rely on other external dependencies, thus, it runs solely on the .NET Common Language Runtime (CLR) [33]. We use F#'s asynchronous workflows [36] in order to implement parallel lifting and IR optimization (see §IV-B).

TABLE III: The number of IR statements before and after applying IR optimization.

	Before Opt.	**After Opt.**	**Reduction Rate**
LIBC Blob	2,297,506	1,906,024	17%
x86 Blob	138,882,883	114,718,934	17%
x86-64 Blob	103,848,026	87,456,016	16%

To make our front-end efficient, we have refactored and heavily optimized B2R2 for several iterations. Specifically, we have removed unnecessary closures and continuations, and also reduced heap allocations as much as possible by using local data structures such as 'struct tuples' [2] to lower pressure on the garbage collector. We also replaced lists with an array whenever we need a random access to the collection, and inlined performance-critical functions. Finally, we have switched from discriminated unions to enums, and removed exceptions in critical execution paths. As a result, the lifting speed of B2R2 on a single core, without using the multi-core parallel lifting method, has become 2.8× faster after our rigorous optimization. The huge performance gain here signifies the importance of engineering effort.

V. Evaluation

In this section, we evaluate B2R2 with respect to the following research questions: (1) Does optimization help simplifying the IR statements? If so, how much slow down do we observe by applying IR optimization? (2) Can we exploit multi-core parallelism to speed up the lifting process?; (3) How much speed up does bigint splitting provide in terms of evaluating IR statements?

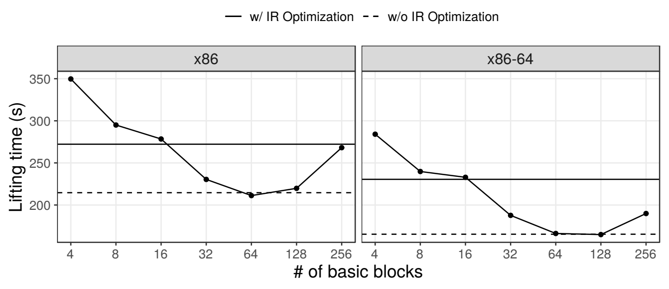
A. IR Optimization

How does IR optimization affect a binary analysis? To answer this question, we first compared the number of IR statements before and after applying our IR optimization. Specifically, we lifted the same target binary blobs used in §III, and applied our IR optimization on the lifted IR statements. As a result, we were able to reduce the number of statements by 16% and 17% in x86 and x86-64, respectively. Table III summarizes the result. Reducing the number of IR statements is beneficial not only for IR evaluation, but also for binary analyses in general. For example, one may expect that IR optimization helps reduce the size of the resulting path formulas.

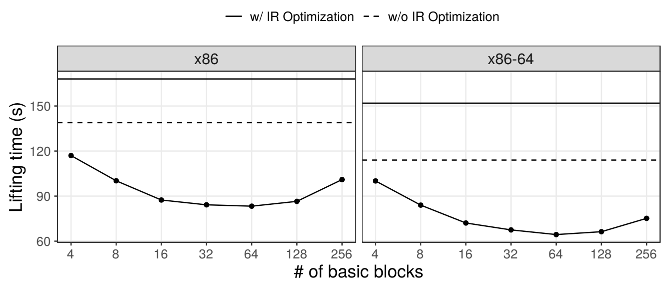
However, IR optimization can cause an extra overhead considering only the lifting process. In our experiments, IR optimization incurred 26.5% overhead on average on our dataset. This result naturally motivates the next research question about exploiting multi-core parallelism (§V-B).

B. Exploiting Multi-core Parallelism

Recall from §IV-B, our parallel lifting technique leverages multiple cores to enhance the performance of the front-end. To measure the effectiveness of parallel lifting, we compared the total lifting time with and without our parallel lifting technique. When running with parallel lifting, we measured time for the whole pipeline of the B2R2 front-end including



(a) On our server machine (Xeon E5-2620 v4).



(b) On our desktop machine (i7-8700).

Fig. 4: Parallel lifting performance comparison with varying number of basic blocks to accumulate.

parsing, lifting, optimization, and pretty-printing. We used two different machines for this experiment: (1) a 6-year-old Linux server machine with Intel Xeon E5-2620 v4 (32 cores); and (2) a modern Windows desktop machine with Intel i7-8700 (12 cores). We chose these machines to measure the impact of parallel lifting on different environments.

On each machine, we changed N, the number of basic blocks to accumulate, from 4 to 256, and measured the overall lifting time (see §IV-B). Figure 4 illustrates the results. The lines with dots represent the lifting time for parallel lifting. As we increase the number of basic blocks to accumulate, we observed better throughput until the number reaches 64. However, when we accumulate more than 64 blocks, we started to observe worse performance because the memory pressure to the garbage collector increases as we accumulate more basic blocks and their ASTs.

In both machines, however, the total lifting time with parallel lifting was less than that without performing any IR optimization. This means our parallel lifting technique can significantly benefit our front-end engine, and it can even make it faster than the speed of lifting without IR optimization. Furthermore, our parallel lifting prevailed in all cases on our desktop machine. It was at most 77% faster than the speed of lifting without IR optimization. This corresponds to the lifting speed of radare2 that we showed in our preliminary study (§III-C). We suspect the reason why we observe significant differences between the two machines is that our server machine has poor memory bandwidth, which is about a half the one of the desktop.

C. Bigint Splitting

Recall from §IV-G, our bigint splitting enables representing the semantics of binaries with fixed-precision integers. We can observe the impact of bigint splitting by evaluating IR statements. To measure such an impact, we took execution traces from GNU coreutils and concretely emulated the executions based on the concrete execution context.

Specifically, we first gathered 73 and 72 coreutils binaries for x86 and x86-64, respectively. We then randomly selected one of their unit tests for each program, and ran the target program with our own execution tracer implemented with Pin [32]. Our tracer runs a target program, and records register and memory values accessed throughout the execution of the program. We omitted several coreutils binaries that are difficult to extract a test case from the unit test scripts. Given an execution trace, our execution emulator lifts instructions stored in the trace into LowUIR, and evaluates the IR statements by using the register and memory values stored in the trace.

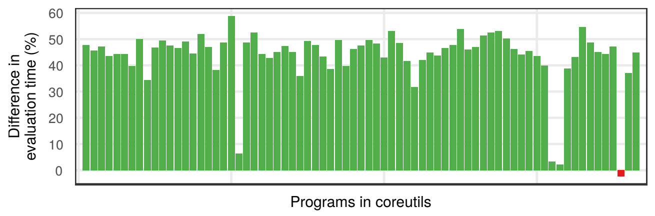
Figure 5 shows the rate between the execution time with and without bigint splitting for each coreutils binary. Out of total 145 binaries, only 135 binaries showed a significant performance improvement. Overall, we observed 43.5% and 8.3% performance gain for x86 and x86-64, respectively. We note that we have substantially more performance gain on x86 because in our experiments, x86 coreutils binaries had more number of complex arithmetic instructions such as multiplication instructions: we observed three times more `imul` instructions on x86 binaries. A multiplication operation with arbitrary-precision integers is significantly slower than the one with fixed-precision integers. On the other hand, bigint splitting can slow down the evaluation of the other 10 binaries as the number of variables to handle increases, which, in turn, slows down the insert and the find operations of our dictionary, storing the mappings from variables to the corresponding values. Our current implementation uses a functional finite map, which internally uses a binary tree to implement the dictionary. However, we can mitigate the performance bottleneck here by employing a hash table.

We conclude that bigint splitting can lead to substantial speed up for IR evaluation, especially when there are instructions involving complex arithmetic operations.

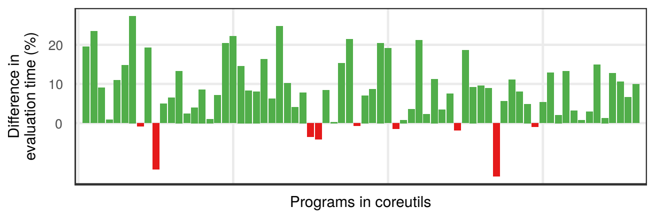
D. Threats to Validity

1) Representativeness of Target Binaries: When the instruction types in the target binaries are limited, our experimental results can be undermined. To mitigate this problem, we tried to collect a large number of binary executables from a real-world OS for two different ISAs: x86 and x86-64. We assumed that `/bin` and `/usr/bin` contain a reasonable amount of binaries with various functionalities.

2) Correctness of our Source Modification: Recall from §III-B, we have modified the source code of other binary analysis tools. Although we sincerely examined the code, our modification may have unexpected side-effects, which can affect their lifting performance. To alleviate the concern, We



(a) Comparison on the x86 coreutils.



(b) Comparison on the x86-64 coreutils.

Fig. 5: Effectiveness of bigint splitting.

will put the diff files we used on GitHub along with the source code of our system.

VI. B2R2 Capabilities and Applications

Although the primary focus of this paper is on the front-end, B2R2 currently includes various back-end modules too. In this section, we describe two of the back-end modules we have as well as an application that we developed on B2R2.

A. *Return-oriented Programming Compilation*

Building a Return-Oriented Programming (ROP) chain is a crucial part of exploit development. B2R2 employs a ROP compilation module that analyzes the given binary and returns a ROP payload, which is primarily inspired by Q [37]. It first searches for useful ROP gadgets, and combine the gadgets to build a ROP payload for calling an arbitrary function, invoking a system call, spawning a shell, or performing a stack pivoting.

B. *Graph Visualization*

B2R2 not only provides rich features for automatic binary analyses, but it also aids analysts to manually inspect target binaries. In particular, B2R2 has its own graph visualization engine for drawing Control-Flow Graphs (CFGs). Many existing binary analysis frameworks such as BAP, BINSEC, and PyVEX rely on external graph layout tools, which make it difficult to analyze CFGs in an interactive manner. Our visualization algorithm follows the standard hierarchical drawing methods based on Sugiyama framework [42].

C. *Symbolic Execution*

Symbolic execution on binary code [8], [16], [40], [49] is gaining a growing attention from the research community, due to its usefulness in various fields including vulnerability detection, automatic exploit generation and de-obfuscation. On top of B2R2, we implemented a prototype symbolic executor that leverages the useful features of B2R2. For example, using the meta information of expressions of LowUIR (§IV-D), we were able to decide whether a basic block should be evaluated symbolically or not. This feature also makes it possible to implement the *constraint independence* [14] without walking ASTs. We also made use of hash-consed ASTs (§IV-F) with expression memoization to speed up the symbolic engine. Moreover, we observed meaningful performance enhancement by leveraging block-level local optimization (§IV-E) and bigint splitting (§IV-G). Discussing the detailed design and implementation of our symbolic execution engine is beyond the scope of this paper.

VII. Conclusion

In this paper, we studied several design aspects of the front-end of binary analysis. We found that current binary analysis frameworks mostly focus on the ease of analysis, but not on their efficiency. Furthermore, they neglect several critical design points for their front-ends such as exploiting parallelism and using fixed-precision integers, which can substantially affect their efficiency. To demonstrates our claim, we designed and implemented B2R2, a new binary analysis platform that is functional-first, parallelizable, and efficient in both lifting and evaluating IRs. We make our source code public on GitHub.

References

[1] "ESIL: Radare2 book," https://radare.gitbooks.io/radare2book/content/esil.html.

[2] "F# RFC FS-0006—struct tuples and interop with C# 7.0 tuples," https://github.com/fsharp/fslang-design/blob/master/FSharp-4.1/FS-1006-struct-tuples.md.

[3] "PyVEX," https://github.com/angr/pyvex.

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

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[6] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with Veritesting," in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 1083–1094.

[7] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "CodeSurfer/x86—a platform for analyzing x86 executables," in *Proceedings of the International Conference on Compiler Construction*, 2005, pp. 250–254.

[8] S. Bardin, R. David, and J.-Y. Marion, "Backward-bounded dse: Targeting infeasibility questions on obfuscated codes," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2017, pp. 633–651.

[9] S. Bardin and P. Herrmann, "OSMOSE: Automatic structural testing of executables," *Software Testing, Verification & Reliability*, vol. 21, no. 1, pp. 29–54, 2011.

[10] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA framework for binary code analysis," in *Proceedings of the International Conference on Computer Aided Verification*, 2011, pp. 165–170.

[11] D. Beazley, "Inside the Python GIL," http://www.dabeaz.com/python/GIL.pdf.

[12] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of the International Conference on Software Engineering*, 2013, pp. 122–131.

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

[14] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2008, pp. 209–224.

[15] CEA IT Security, "Reverse engineering framework in python," https://github.com/cea-sec/miasm.

[16] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.

[17] A. Di Federico, M. Payer, and G. Agosta, "Rev.ng: A unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the International Conference on Compiler Construction*, 2017, pp. 131–141.

[18] A. Djoudi and S. Bardin, "BINSEC: Binary code analysis with low-level regions," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 212–217.

[19] T. Dullien and S. Porst, "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," in *Proceedings of the CanSecWest*, 2009.

[20] J.-C. Filliâtre and S. Conchon, "Type-safe modular hash-consing," in *Proceedings of the Workshop on ML*, 2006, pp. 12–19.

[21] E. Fleury, O. Ly, G. Point, and A. Vincent, "Insight: An open binary analysis framework," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 218–224.

[22] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[23] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from i/o samples," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2012, pp. 441–452.

[24] I. Gotovchits, R. van Tonder, and D. Brumley, "Saluki: Finding taint-style vulnerabilities with static property checking," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2018.

[25] I. Guilfanov, "Hex-rays decompiler internals: Microcode," https://hex-rays.com/products/ida/support/ppt/recon2018.ppt.

[26] N. Hasabnis and R. Sekar, "Lifting assembly to intermediate representation: A novel approach leveraging compilers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 311–324.

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

[28] INRIA, "Multicore OCaml," https://github.com/ocamllabs/ocaml-multicore/.

[29] S. Kim, M. Faerevaag, M. Jung, S. J. D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *Proceedings of the International Conference on Automated Software Engineering*, 2017, pp. 353–364.

[30] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proceedings of the International Conference on Computer Aided Verification*, 2008, pp. 423–427.

[31] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium*, 2011, pp. 251–268.

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.

[33] Microsoft, "Common language runtime (CLR) overview." https://docs.microsoft.com/en-us/dotnet/standard/clr.

[34] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2007, pp. 89–100.

[35] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[36] T. Petricek and D. Syme, "The F# computation expression zoo," in *Proceedings of Practical Aspects of Declarative Languages*, 2014, pp. 33–48.

[37] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proceedings of the USENIX Security Symposium*, 2011, pp. 25–41.

[38] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, "Using logic programming to recover C++ classes and methods from compiled executables," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018, pp. 426–441.

[39] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proceedings of the USENIX Security Symposium*, 2013, pp. 353–368.

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

[41] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the International Conference on Information Systems Security*, 2008, pp. 1–25.

[42] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[43] D. Syme, G. Neverov, and J. Margetson, "Extensible pattern matching via a lightweight language extension," in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2007, pp. 29–40.

[44] N. Tillmann and J. De Halleux, "Pex–white box test generation for .NET," in *Proceedings of the International Conference on Tests and Proofs*, 2008, pp. 134–153.

[45] Trail of Bits, "McSema," https://github.com/trailofbits/mcsema.

[46] R. van Tonder and C. Le Goues, "Cross-architecture lifter synthesis," in *Proceedings of the International Conference on Software Engineering and Formal Methods*, 2018, pp. 155–170.

[47] Vector 35, "Binary Ninja," https://binary.ninja/.

[48] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[49] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the USENIX Security Symposium*, 2018, pp. 745–762.