A Broad Comparative Evaluation of x86-64 Binary Rewriters

Eric Schulte
schulte.eric@gmail.com

Michael D. Brown
Trail of Bits
New York, NY, USA
michael.brown@trailofbits.com

Vlad Folts
Grammatech, Inc.
Ithaca, NY, USA
vfolts@grammatech.com

ABSTRACT

Binary rewriting is a rapidly-maturing technique for modifying software for instrumentation, customization, optimization, and hardening without access to source code. Unfortunately, the practical applications of binary rewriting tools are often unclear to users because their limitations are glossed over in the literature. This, among other challenges, has prohibited the widespread adoption of these tools. To address this shortcoming, we collect ten popular binary rewriters and assess their *generality* across a broad range of input binary classes and the functional *reliability* of the resulting rewritten binaries. Additionally, we evaluate the *performance* of the rewriters themselves as well as the rewritten binaries they produce.

The goal of this broad evaluation is to establish a shared context for future research and development of binary rewriting tools by providing a *state of the practice* for their capabilities. To support potential binary rewriter users, we also identify input binary features that are predictive of tool success and show that a simple decision tree model can accurately predict whether a particular tool can rewrite a target binary. The binary rewriters, our corpus of 3344 sample binaries, and the evaluation infrastructure itself are all freely available as open-source software.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**.

KEYWORDS

Binary Rewriting, Binary Analysis, Binary Recompilation

ACM Reference Format:
Eric Schulte, Michael D. Brown, and Vlad Folts. 2022. A Broad Comparative Evaluation of x86-64 Binary Rewriters. In *Cyber Security Experimentation and Test Workshop (CSET 2022), August 8, 2022, Virtual, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3546096.3546112

1 INTRODUCTION

Binary rewriting (also referred to as recompilation) is an emerging research area that has been enabled by recent advances in binary analysis. Binary rewriting tools have the potential to address longstanding problems in cyber security by enabling binary analysis, patching, and security hardening for programs where source code is not available (e.g., legacy or closed-source software). For binary rewriting tools to be viable, they must *generalize* to the full variety of programs available on heterogeneous computing platforms and *reliably* produce functional rewritten binaries.

A surfeit of research into binary rewriting applications including instrumentation, optimization, configuration, debloating, and hardening reveals a wide and largely under-emphasized variance in the generality and reliability of binary rewriters [6, 8, 9, 16, 24, 25, 27, 29, 30, 34, 37, 42, 45, 46]. Frequently, papers presenting these tools only briefly mention large gaps in generality such as support limited to binaries with relocation information and/or symbols – neither are typical of commercial off-the-shelf (COTS) software [8, 42].

While there has been significant research to date seeking to systematize knowledge of general binary rewriting techniques [38] and evaluate the quality of binary lifters and disassemblers [2, 20], no systematic comparative evaluation of binary rewriters has yet been conducted. In this work, we address this important knowledge gap by conducting such an evaluation of 10 binary rewriters across 3344 variant binaries sourced from 34 benchmark programs and 3 production compilers. The tools evaluated in this work are *static* binary rewriting tools; we exclude dynamic binary rewriting tools (e.g., PIN [19]) whose runtime harnesses and overhead often make them impractical for the applications considered by this work.

Our work differs from previous surveys in two key ways. First, prior work systematizing knowledge on binary rewriters [38] focused primarily on their underlying techniques and algorithms and as such did not evaluate their artifacts empirically. In contrast, our evaluation focuses on measuring and comparing the *generality* and *reliability* of a broad collection of publicly available binary lifter and rewriter tools.[1] Second, prior works performing comparative evaluation of binary disassmblers and lifters [2, 20] focus on *depth* achieving near complete measurement of binary analysis accuracy across a small pool of binaries. The difficulties implicit in a truly thorough analysis limits the breadth of these works to small numbers of binaries or to specific classes of binaries. Our evaluation focuses on input program *breadth* to directly address tool *generality* and rewritten binary functionality to directly address *reliability*.

Summary of Contributions. In this paper, we first review related work evaluating binary transformation tools in Section 2. We then describe our experimental methodology to assess the generality and reliability of existing tools in Section 3. Next, we present our experimental results and predictive models derived from them in Section 4. Finally, we discuss the state of practice in binary rewriting tools and options for potential users in Section 5.

2 BACKGROUND

2.1 Types of Binary Rewriters

Direct Rewriting. Zipr [14] lifts a binary into an Intermediate Representation Data Base (IRDB [18]) upon which transformations

CSET 2022, August 8, 2022, Virtual, CA, USA

ACM ISBN 978-1-4503-9684-4/22/08…$15.00
https://doi.org/10.1145/3546096.3546112

[1] We are not aware of any comparable closed source binary rewriters.

may be applied. The IRDB can then be written directly to a binary executable. Similarly, Egalito [42] lifts a binary into a low-level Intermediate Representation (IR), provides an API for transforming this IR, and supports lowering this IR back to a modified binary.

Reassemblable Disassemblers. Uroboros [36] – superseded by the Phoenix reassemblable decompiler [23] – popularized the idea of reassemblable disassembly (i.e., disassembly to assembly code that can be readily reassembled). Retrowrite [8] also emits reassemblable assembly code. DDisasm [10] lifts to GrammaTech's Intermediate Representation for Binaries (GTIRB) [26], and can also directly emit assembly code that can be reassembled.

LLVM Rewriting. The now defunct SecondWrite [29] was the first tool to lift binaries to LLVM IR. More recently, McSema [7] and rev.ng [6] have become the predominant binary lifters to target LLVM IR. Although LLVM has a large user community and provides many analysis, optimization, and hardening passes, there are two properties of its IR that make it difficult for lifters to target. First, it requires *type* information to represent data. This requires binary type analysis, which is prohibitively difficult at the scale required to rewrite large programs. Instead, many tools explicitly *emulate* the stack, stack maintenance around function calls, and memory as a large array of bytes. The lack of true types and stack/memory emulation limits the utility of most existing LLVM passes and results in baroque and inefficient rewritten binaries [15]. Second, LLVM represents code in static single assignment form resulting in a loss of specificity with respect to the original instructions.

Trampoline Rewriting. Trampoline rewriters such as e9patch minimally disturb the memory image of the original binary [4, 9]. New code is placed outside of the original image and the only changes within the image are jumps to this new code, which itself jumps back to the original code. Trampoline rewriting can be very robust as it requires minimal binary analysis (e.g., symbolization is not necessary), however it is only well-suited to *additive* transformation. The original code cannot be modified, moved, optimized, or removed easily or effectively.

2.2 Related Work

Wenzl et. al. [38] survey over 50 years of binary rewriter research with the primary objective of categorizing binary rewriting approaches by end use-case, analysis techniques, code transformation methods, and code generation methods. However, this survey does not include a comparative evaluation of the tools presented in the literature. Our work aims to complement and extend their survey by providing an empirical evaluation of binary rewriters.

Several works performing extensive evaluations of binary disassemblers have been published in recent years [2, 17, 20, 22]. Collectively, these works thoroughly document the approaches, challenges, trade-offs, and shortcomings of disassemblers. Further, they establish that modern disassemblers achieve high accuracy (close to 100%) even in the presence of challenging (although rare) code constructs due to advances and specialization in code discovery and control-flow reconstruction algorithms. These evaluations are similar in scale to our work, however we evaluate tools with custom disassembly routines not covered in these works (all except uroboros). Further, our evaluation focuses on binary rewriting as opposed to disassembly, which is a prerequisite step to rewriting.

Additionally, several evaluation frameworks and benchmark suites for evaluating binary analysis tools related to binary rewriting have been recently proposed. Mishegos [43] is a novel differential fuzzer that can be used to evaluate x86-64 instruction decoders. ConFIRM [44] is an evaluation framework and benchmark suite for assessing software hardening transformations, namely control-flow integrity implementations. Finally, Dasgupta et. al. [5] recently presented a scalable technique for validating and detecting bugs in binary lifters such as McSema [7].

3 METHODOLOGY

3.1 Tool Selection

We selected ten binary rewriters for our evaluation, listed in Table 1. While our corpus of tools is not exhaustive, it provides excellent coverage of tools that are mature, robust, and scale via automation. We exclude two notable binary rewriting tools, McSema[7] and Ramblr[35], from our evaluation.

In McSema's case, the tool can be automated to lift a wide variety of binaries to LLVM IR. In our initial evaluation, McSema successfully lifted 57% of the 3344 program variants we tested it against. However, McSema's rewriting workflow currently requires manual steps by a knowledgeable operator. As a result, rewriting binaries with McSema does not satisfy the requirements of our evaluation. While still maintained, McSema's shortcomings in this area are largely due to age (its last major release was in 2018) and the recent modernization of its dependent libraries, Anvill [31] and Remill [33], to support newer lifting tools such as Rellic [32].

In Ramblr's case, it is implemented as a supplementary analysis for the Angr binary analysis framework [28]. In its current from, Ramblr is capable of reassembling x86 and x86_64 binaries disassembled by Angr; however, it does not expose an interface for transforming disassembled binaries. As such, Ramblr cannot be truly used as a binary rewriter (i.e., it cannot perform our second evaluation task) "out of the box".

Table 1: Tools selected for this evaluation

Tool	Rewriter Type
ddisasm [10]	Reassemblable Disassembler
e9patch [9]	LLVM Trampoline
Egalito [42]	Direct Rewriter
mctoll [21]	LLVM Rewriter
multiverse [3]	Direct Rewriter
ReOpt [11]	LLVM Rewriter
revng [6]	LLVM Rewriter
Retrowrite [8]	Reassemblable Disassembler
Uroboros [36]	Reassemblable Disassembler
Zipr [14]	Direct Rewriter

3.2 Evaluation Variant Generation

In order to obtain realistic evaluation results, we combined benchmark lists compiled by two program managers from the United States Department of Defense to arrive at a diverse list of 34 real-world benchmark programs, shown in Table 2. To this corpus we also added a trivial "Hello World!" program to our corpus to provide a low-water mark for program complexity. For each benchmark and the hello-world program, we compiled an x86-64 variant of the program using one permutation of the compilers, optimization levels, code layout, symbol options, and operating systems listed in Table 3. In total, we generated 3344 distinct evaluation variants.

3.3 Evaluation Tasks

We evaluate our selected binary rewriters based on their ability to successfully perform two rewriting *tasks* and record their progress at multiple *checkpoints*. In the interest of breadth we use a proxy for successful (i.e., correct) rewriting. Specifically, we consider a rewrite to be successful if the output executable passes a very simple functional test (described in subsection 3.4). In practice many binary rewriting tools fail fast when problems arise, meaning they either completely fail to produce a new executable or they produce a executable that is unable to start execution. For a small subset of our benchmark with readily executable test suites with high coverage we also tested programs against the full test suite.

Tasks. The two tasks we use to evaluate our tools are:

NOP This task is a minimal NOP (i.e., No Operation) transform that simply lifts the binary and then rewrites without modification. The NOP transform tests the ability of a binary rewriter to successfully process the input binary, populate its internal or external intermediate representation of the binary, and then produce a new rewritten executable. Despite its name this transform is decidedly non-trivial for most rewriters, evidenced by the fact that rewritten binaries typically look very different from the original.

AFL This task is a more complex transform characteristic of the needs of instrumentation, e.g. to support gray-box fuzz testing. It evaluates our tools' abilities to extensively *transform* a binary with instrumentation to support AFL++ [1][2]. This task is important to include as many rewriters cover up analysis errors by incorporating reasonable defaults (e.g., linking code from the original binary on lifting failure, or preserving unidentified symbols which continue to resolve correctly if code is left undisturbed in memory).

Checkpoints. For every attempted rewrite operation, we collect some of the following artifacts to checkpoint the process:

IR For every binary rewriting tool that leverages some form of external IR, we collect that IR. Specifically, we collect the ASM files generated by tools that emit reassemblable disassembly and the LLVM IR for tools targeting LLVM. Zipr, Egalito, and multiverse use a internal IRs that are not easily serialized to disk. E9patch does not present the original code for rewriting. As such, we do not track successful IR generation for these tools.

EXE We next check if the rewriter successfully creates a new executable. In some cases rewritten executables are trivially recreated and are not an indicator of success (e.g., Egalito almost always generates a new executable even if most of them are non-functional). However, in most cases the ability to re-assemble and re-link a new executable indicates that the rewriting tool both successfully disassembled reasonable assembly instructions and generated the required combinations of symbols and included libraries.

3.4 Evaluation Metrics

Functional Metrics. To measure rewriter correctness, we first observe the success rate for each tool across all variants for both tasks (i.e., NOP and AFL) at both checkpoints (i.e., IR and EXE). Next, we perform a simple invocation of the NOP rewritten programs (e.g., running `--help`) to ensure the rewrite did not obviously corrupt the program. We refer to this as the *Null Function* test. Finally, we execute the AFL rewritten programs with the driver provided by AFL++ to ensure instrumenting the program via binary rewriting did not corrupt the program and that instrumentation was successfully incorporated. We refer to this as the *AFL Function* test.

Non-Functional Metrics. To measure rewriter runtime performance we observe the total required runtime and the memory high-water mark used by tools during rewriting. Available memory is often the limiting factor for rewriting because many underlying analyses scale super-linearly in the input binary size.

To determine the performance impacts of rewriting on binaries, we first measure file size impacts using Bloaty [12]. Size is an important metric as it measures the degree to which a rewriting tool has inserted dynamic emulation or runtime supports – with their associated increased complexity and runtime costs. Finally, for successfully rewritten program variants with publicly available test suites we measure the impact of rewriting on performance. Specifically, we measure pass rate for all tests in the suite, runtime of the test suite, and the memory consumption high-water mark during execution of the full test suite as compared to the original.

4 EXPERIMENTAL RESULTS

We present binary rewriter success both in aggregate across our entire benchmark set and broken out into cohorts. Each cohort of binaries has like characteristics that highlight the comparative strengths and weaknesses of the evaluated tools.

4.1 Aggregate Rewriting Success Rates

Our aggregate success results are presented in Table 4. Overall, we observed a very broad range of success rates (and by extension levels of support) achieved by our selected binary rewriting tools. For the NOP transform, the tools fall into four distinct categories characterized by the fraction of the universe of potential binaries they can handle:

1. Tools that work only on a tiny fraction (≤5%) of binaries. This group includes mctoll and uroboros.
2. Tools which work on a few (∼10%) binaries. This group includes multiverse and retrowrite.
3. Tools that work on some (∼33%) binaries. This group includes egalito, reopt, and revng.
4. Tools that work on most (≥80%) binaries. This group includes ddisasm, e9patch, and zipr.

In category (1) we find tools that handle a very limited set of binaries (e.g., mctoll only successfully transformed hello-world binaries). The tools in category (2) support a wider range of binaries but in many cases make hard and fast assumptions (e.g., multiverse can only successfully rewrite binaries compiled by old versions of

[2] Each tool we selected except multiverse claims to support AFL++'s instrumentation.

Table 2: Benchmark programs used in this evaluation

Program	SLOC	Description	Program	SLOC	Description
anope	65,441	IRC Services	openvpn	89,312	VPN Client
asterisk	771,247	Communication Framework	pidgin	259,398	Chat Client
bind	376,147	DNS System	pks	40,788	Public Key Server
bitcoind	229,928	Bitcoin Client	poppler	188,156	PDF Reader
dnsmasq	34,671	Network Services	postfix	134,957	Mail Server
filezilla	176,324	FTP Client and Server	proftpd	544,178	FTP Server
gnome-calculator	301	Calculator	qmail	14,685	Message Transfer Agent
leafnode	12,945	NNTP Proxy	redis	14,685	In-memory Data Store
Libreoffice	5,090,852	Office Suite	samba	1,863,980	Windows Interoperability
libzmq	62,442	Messaging Library	sendmail	104,450	Mail Server
lighttpd	89,668	Web Server	sipwitch	17,134	VoIP Server
memcached	33,533	In-memory Object Cache	snort	344,877	Intrusion Prevention
monerod	394,783	Blockchain Daemon	sqlite	292,398	SQL Server
mosh	12,890	Mobile Shell	squid	212,848	Caching Web Proxy
mysql	3,331,683	SQL Server	unrealircd	90,988	IRC Server
nginx	170,602	Web Server	vi/vim	394,056	Text Editor
ssh	127,363	SSH Client and Server	zip	54,390	Compression Utility

Table 3: Variant configuration options

Compiler	Flags	Relocation (Position-)	Symbols	Operating Systems
clang	O0	Independent	Present	Ubuntu 16.04[1]
gcc	O1	Dependent	Stripped	Ubuntu 20.04
icx	O2			
	O3			
	Os			
	Ofast			
OLLVM	fla	Independent	Present	Ubuntu 20.04
	sub	Dependent	Stripped	
	bcf[2]			

[1] Some binaries could not be built on this OS due to unavailable dependencies.

[2] Probability variable set to always insert (100%)

clang and gcc available on Ubuntu 16.04). These tools still do not handle binaries that make use of fairly common code structures (e.g., C++ exceptions). The tools in category (3) largely only work with relocation and debug information, but are able to handle a wide range of the binaries meeting these restrictions. Finally, category (4) tools do *not* require relocation or debug information and support a wide range of both complex code structures and compiler-specific binary generation behaviors such as multiple forms of jump tables, data intermixed with code, and specialized control-flow constructs.

Given that so many tools require relocation and debug information we present a second view of our results limited to binaries that include this information (i.e., non-stripped, position-independent variants) in the right of Table 4. Although position-independent binaries are increasingly common as ASLR becomes the norm, it is still uncommon for COTS binaries to include debug information.

These results show the increase in rewriting success rate that a developer might expect if they compile their binaries with relocation and debug information to support binary rewriting. However, such binaries are not characteristic of the stripped COTS binaries likely to be received from third parties or found in the wild.

As shown in Table 4, the AFL transform provides a much better proxy for actual binary rewriter performance than the NOP transform. This is true for at least two reasons. First, when applying the NOP transform many relative and absolute locations in a rewritten binary will continue to match their locations in the original binary because no attempt is made to modify the lifted code. This provides a great deal of grace for rewriters that missed code or symbols in the original binary because symbols treated as literals or as data in NOP transformed binaries remains sound surprisingly often. Second, the AFL test is stricter because the rewritten binary must successfully interact with the AFL++ harness to record a successful execution.

Every rewriter included in this evaluation except multiverse provides some out-of-the-box support for AFL++ instrumentation. ddisasm, retrowrite, and uroboros all produce assembly-level IR that an AFL++ provided tool can directly instrument. Similarly mctoll, reopt, and revng produce LLVM IR that an AFL++ provided LLVM pass can directly instrument. E9patch, egalito, and zipr ship with an AFL++ instrumentation pass compatible with their frameworks.

Despite this broad support, only ddisasm, mctoll, retrowrite, and zipr successfully transform *any* of our test binaries for use with AFL++. In Egalito's case, the included AFL++ transform requires a patched `afl-fuzz` program [39–41]. Reopt and revng appear to produce LLVM IR that is not suitable for transformation. Further, it appears reopt may only perform well on the NOP transform because it falls back to directly re-linking sections of the original binary when rewriting fails. Uroboros appears to fail to produce any functional AFL transformed binaries not due to any uniform systematic reason but simply because the rewritten assembly code is very brittle due to incorrect analyses during lifting.

4.2 Rewriting Success Rates by Compiler

In this section we present the success rates of our binary rewriters broken out by the compiler used to generate variants. Binary

Table 4: Number and percentage of x86-64 Linux binaries for which the rewriter successfully produces IR, produces a NOP-transformed executable ("EXE"), passes the Null Function test, produces an AFL++ instrumented executable ("AFL EXE"), and passes the AFL Function test. Data is aggregated across the the full suite of binaries (3344) in the first set of columns and across position-independent, non-stripped binaries (1672) in the second set.

	Full Suite					**Position-Independent, Symbols Present only**				
Tool	IR	EXE	Null Func.	AFL EXE	AFL Func.	IR	EXE	Null Func.	AFL EXE	AFL Func.
ddisasm	3282	2972	2873	3020	2346	1638	1428	1385	1456	1125
%	98.14%	88.87%	85.91%	90.31%	70.15%	97.96%	85.40%	82.82%	87.08%	67.28%
e9patch	NA	3344	2620	3344	1212	NA	1672	1310	1672	607
%	NA	100%	78.34%	100%	36.24%	NA	100%	78.34%	100%	36.30%
egalito	NA	3294	983	2493	0	NA	1661	492	1261	0
%	NA	98.50%	29.39%	74.55%	0	NA	99.34%	29.42%	75.40%	0
mctoll	30	30	30	30	30	30	30	30	30	30
%	.89%	.89%	.89%	.89%	.89%	1.78%	1.78%	1.78%	1.78%	1.78%
multiverse	NA	880	362	0	0	NA	437	181	0	0
%	NA	26.31%	10.82%	0	0	NA	26.12%	10.82%	0	0
reopt	3007	2556	1134	0	0	1504	1425	684	0	0
%	89.92%	76.43%	33.91%	0	0	89.94%	85.22%	40.90%	0	0
retrowrite	817	334	309	330	254	817	334	309	330	254
%	24.43%	9.98%	9.24%	9.86%	7.59%	48.86%	19.96%	18.48%	19.72%	15.18%
revng	NA	885	786	0	0	NA	439	390	0	0
%	NA	26.46%	23.50%	0	0	NA	26.26%	23.32%	0	0
uroboros	364	216	96	210	0	183	108	48	105	0
%	10.88%	6.45%	2.87%	6.27%	0	5.47%	6.44%	2.86%	6.26%	0
zipr	NA	3344	3344	2708	1609	NA	1672	1672	1560	1079
%	NA	100%	100%	80.98%	48.11%	NA	100%	100%	93.30%	64.52%

rewriting success is often dependent on the compiler used to produce the input binary as many of the heuristics baked into rewriting tools target binary code generation logic or optimizations specific to certain compilers. For example, the Intel compiler (`icx`) in-lines data into the code section on Linux whereas Clang and GCC do not. As a result of this behavior and other ICX-specific optimizations, some binary rewriters have a significantly lower success rate against ICX-produced binaries.

The results restricted to GCC-compiled variants in Table 5 are most similar to the aggregate results. This is unsurprising as GCC is the prototypical compiler for Linux systems and represents a middle ground in optimization aggressiveness between the relatively conservative Clang and the very aggressive Intel ICX. Interestingly, the success rate for Clang-compiled variants (Table 5) across all tools is slightly higher than GCC's success rate. This could be due to a number of factors including GCC-only optimizations that prove difficult for binary rewriting tools or GCC leveraging non-standard ELF file format extensions that are not produced by Clang.

Our ICX-compiled results are shown in Table 6. They vary widely from both GCC and Clang and also across our evaluated tools. DDisasm performs better on ICX binaries generating an IR in 99% of cases and generating functional AFL rewrites 2% more frequently with these binaries than in aggregate. By contrast, Egalito's NOP transform success rate drops for ICX-produced variants, mctoll, Multiverse, and Uroboros are unable to process any ICX binaries, and Retrowrite and Zipr both perform significantly worse on ICX binaries although they are still able to successfully generate functional AFL++ instrumented binaries in some cases.

Given the obfuscation techniques employed by ollvm are meant to hinder binary analysis and reverse engineering, we expected the tools to perform worst against ollvm compiled binaries (Table 6). However, we were surprised to find that in most cases the rewriting success rate increased for these binaries. It is not clear if this is because those programs which could be compiled with ollvm represent the simpler end of our benchmark set, or if there is something about the ollvm transformations that are amenable to binary rewriting if not to traditional reverse engineering.

4.3 Analysis of Binary Rewriter Success

In this section, we investigate binary formatting options to determine if they are correlates for binary rewriter success. We collected three readily-identifiable formatting features using `readelf` from GNU `binutils`: (1) whether or not the binary is position-independent, (2) whether or not the binary is stripped, and (3) the sections included in the binary[3].

We collated the success and failure rate across these features for each tool against our corpus of variants considering both rewriting tasks (i.e., NOP and AFL). Then we identified the four most predictive features for rewriting success or failure of the AFL transform for each tool. These features are presented in Table 7. In many cases these features are expected and match the advertised capabilities

[3] We exclude sections which appear in all binaries (e.g., .text) and sections unique to specific program (e.g., .gresource.gnome_calculator).

Table 5: Number and percentage of 1280 GCC-compiled and 1176 Clang-compiled x86-64 Linux binaries for which the rewriter successfully reaches task checkpoints and passes functional tests.

	GCC					Clang				
Tool	IR	EXE	Null Func.	AFL EXE	AFL Func.	IR	EXE	Null Func.	AFL EXE	AFL Func.
ddisasm	1237	1146	1106	1157	871	1167	1034	990	1057	799
%	96.64%	89.53%	86.40%	90.39%	68.04%	100%	87.93%	84.18%	89.88%	67.94%
e9patch	NA	1280	994	1280	453	NA	1174	952	1174	442
%	NA	100%	77.65%	100%	35.39%	NA	99.83%	80.95%	99.83%	37.59%
egalito	NA	1257	235	1113	0	NA	1161	494	1052	0
%	NA	98.20%	18.36%	86.95%	0	NA	98.72%	42.01%	89.46%	0
mctoll	16	16	16	16	16	11	11	11	11	11
%	1.25%	1.25%	1.25%	1.25%	1.25%	.94%	.94%	.94%	.94%	.94%
multiverse	NA	438	176	0	0	NA	442	186	0	0
%	NA	34.22%	13.75%	0	0	NA	37.59%	15.82%	0	0
reopt	1168	874	387	0	0	1057	901	463	0	0
%	91.25%	68.28%	30.23%	0	0	89.88%	76.62%	39.37%	0	0
retrowrite	308	129	124	126	99	288	147	136	146	108
%	24.06%	10.08%	9.69%	9.84%	7.73%	24.49%	12.50%	11.56%	12.41%	9.18%
revng	NA	394	354	0	0	NA	379	340	0	0
%	NA	30.78%	27.66%	0	0	NA	32.23%	28.91%	0	0
uroboros	150	92	16	92	0	144	92	56	86	0
%	11.72%	7.19%	1.25%	7.19%	0	12.24%	7.82%	4.76%	7.31%	0
zipr	NA	1280	1280	1033	674	NA	1174	1174	1010	671
%	NA	100%	100%	80.70%	52.66%	NA	99.83%	99.83%	85.88%	57.06%

Table 6: Number and percentage of 646 ICX-compiled and 244 Ollvm compiled and obfuscated x86-64 Linux binaries for which the rewriter successfully reaches task checkpoints and passes functional tests.

	ICX					OLLVM				
Tool	IR	EXE	Null Func.	AFL EXE	AFL Func.	IR	EXE	Null Func.	AFL EXE	AFL Func.
ddisasm	641	561	552	574	468	237	231	225	232	208
%	99.23%	86.84%	85.44%	88.85%	72.45%	97.13%	94.67%	92.21%	95.08%	85.25%
e9patch	NA	646	466	646	214	NA	244	208	244	103
%	NA	100%	72.14%	100%	33.13%	NA	100%	85.24%	100%	42.21%
egalito	NA	636	136	119	0	NA	240	118	209	0
%	NA	98.45%	21.05%	18.42%	0	NA	98.36%	48.36%	85.66%	0
mctoll	0	0	0	0	0	3	3	3	3	3
%	0	0	0	0	0	1.23%	1.23%	1.23%	1.23%	1.23%
multiverse	NA	0	0	0	0	NA	0	0	0	0
%	NA	0	0	0	0	NA	0	0	0	0
reopt	575	571	173	0	0	207	210	111	0	0
%	89.01%	88.39%	26.78%	0	0	84.84%	86.07%	45.49%	0	0
retrowrite	157	24	23	24	18	64	34	26	34	29
%	24.30%	3.71%	3.56%	3.71%	2.79%	26.23%	13.93%	10.66%	13.93%	11.89%
revng	NA	24	16	0	0	NA	88	76	0	0
%	NA	3.71%	2.47%	0	0	NA	36.07%	31.15%	0	0
uroboros	4	0	0	0	0	66	32	24	32	0
%	.62%	0	0	0	0	27.05%	13.11%	9.84%	13.11%	0
zipr	NA	646	646	479	154	NA	244	244	186	110
%	NA	100%	100%	74.15%	28.84%	NA	100%	100%	76.23%	45.08%

of each tool. For example, retrowrite only supports relocatable (i.e., `pi`) and non-stripped (i.e., `strip`) binaries which are its two most predictive features for success.

Next we train a decision tree based on this feature collection to predict the likelihood of success of each rewriter against an example binary when using the AFL transform. Before training we use linear support vector classification to select the most discriminating features for that rewriter. The resulting decision trees are printed as Python code in Appendix B. We evaluate the resulting decision tree using 70% of our binaries for training and reserving 30% for testing. The accuracy of the resulting tree is shown in Table 7.

As shown in Table 7 the resulting decision trees, despite their reliance on very simple binary features were very accurate in predicting the chances of tool success. We anticipate two benefits from this analysis. First, tool developers will have insight into properties of binaries that cause their rewriting tools to fail. Second, users can nearly instantaneously run a combination of `readelf` and our decision tree to see what tools, if any, will reliably transform a given target binary. This is useful when many binary rewriting tools can run for minutes and even hours on a single binary. The success of this simple machine learning model trained on simple inputs indicates promising new directions for the practical application of binary rewriting technology discussed in Section 5. The decision trees and the code used to build and train them are included in our publicly available artifact repository.

4.4 Size of Rewritten Binaries

Changes in binary size (shown in column 3 of Table 8) reflect a tool's design decisions and can impact the utility, efficiency, and potential use cases for the rewritten binary. On average, ddisasm and retrowrite's rewritten binaries are slightly smaller, likely due to symbol and debug information dropped during their rewriting processes. For Egalito, mctoll, uroboros, and zipr the size of the binary increases by a non-trivial amount. Revng and Multiverse are outliers with rewritten binaries that are nearly 16 and 9 times the size of the original, respectively. For Revng, the recompiled binaries exhibit very large increases in unmapped areas of the binary (223 times on average!), potentially indicating flaws in the recompilation stage that can be rectified. For Multiverse the increase is due to a defining design decision: it produces rewritten binaries that contain all *possible* disassemblies of the original binary.

It is important to note that size increases are calculated for binaries the tool can successfully rewrite, which varies for each tool. As a result, a direct comparison of tools using Table 8 is not possible. As such, we also conducted a comparative evaluation of binary size increases between pairs of tools where each successfully rewrote one or more of the same programs (data shown in Appendix A). The best performing tools were retrowrite and ddisasm which produced binaries that were on average 62% and 65% the size of those produced by the other tools, respectively. The tools that produced the largest binaries were revng and multiverse, which produced binaries that were approximately 13 and 7 times the size of those produced by the other tools, respectively.

To investigate these size changes further, we analyzed size changes per section per rewriting tool (data shown in Appendix A). Note that in many cases rewriting tools break elf section tables. In these cases bloaty [12], the tool we use to collect section size, is unable to determine sizes for that section in corresponding binaries. In nearly every rewriting tool the largest increase in size of the elf file is in unmapped bytes or bytes that are not accounted for by the section table. This is likely due to at least the following two factors. First, because any extra non-standard runtime harnesses or extra rewriting-specific supports are not properly entered into the section table of the rewritten binary. Second, binary rewriting tools are not penalized for dropping sections or breaking parts of the section header table that are not required for execution.

4.5 Binary Rewriter Performance

To accurately and successfully rewrite a binary executable requires significant static analysis. These analyses often scale super-linearly with the size of the program being rewritten. We summarize the average run time of each tool in Table 8.

As with rewritten program size, the reported averages are skewed because they are calculated across the set of binaries successfully rewritten by each tool. Thus, rewriters that successfully rewrite larger and more complicated binaries have an average that skews higher. To account for this we also present the comparative average in Table 9. In each cell, the comparative average tool runtime across successfully rewritten binaries by both tools is expressed as a percentage of the row tool to the column tool. For example, uroboros runtime is roughly 14.49% of the runtime of ddisasm. A significant trend observable in Table 9 is that tools with higher success rates tend to run longer than tools with low rewriting success rates. This is not surprising as successful tools perform more analyses and more detailed analyses. They also explicitly handle more portions of the ELF file and more edge cases.

4.6 Binary Rewriter Memory High-Water Mark

Like runtime, a tool's memory requirements may make rewriting impractical. For large binaries, memory requirements will frequently outstrip the memory available on a server class machine. We present the average memory high-water mark during binary rewriting in column 2 of Table 8 and for the same reasons described in subsection 4.4 we also conducted a comparative evaluation of memory high-water marks (data shown in Appendix A). As with rewriter performance, we generally find that successful tools require more resources during rewriting.

4.7 Functionality Against Full Test Suite

For three benchmark programs with readily available test suites we check the degree to which successful execution of the Null Function test predicts successful execution of the complete test suite[4]. Our results are presented in Table 10. In each cell we report the number of binaries that completely pass the full test suite and the number of binaries that pass the Null Function test as “full/null”. We report this for each binary rewriter as well as for the original input binaries files. There are up to 60 binaries for each program due to the multiple build options (e.g., compiler, optimization level, pie, stripped, etc.). We do not include ICX-compiled binaries due to

[4] We report full results for only three benchmark programs due to the dearth of high-quality test suites for real-world programs and the high level of effort required for properly configuring them.

Table 7: Decision tree accuracy predicting binary rewriting success based on simple binary features

Rewriter	NOP	AFL	Most Predictive Features (AFL)
ddisasm	90.03%	81.47%	`note.abi-tag`, `interp`, `gcc_except_table` `debug_str`
e9patch	80.57%	86.06%	`pi`, `got.plt`, `data.rel.ro`, `plt.got`
egalito	87.15%		
mctoll	98.80%	98.80%	`strip`, `data.rel.ro`, `symtab`, `strtab`
multiverse	97.80%		
reopt	67.82%		
retrowrite	94.32%	93.02%	`pi`, `strip`, `symtab`, `strtab`
revng	78.78%		
uroboros	96.31%		
zipr	86.65%	79.98%	`strip`, `note.gnu.build-id`, `symtab`, `strtab`

Table 8: Average tool runtime memory high-water mark, and relative program size change across successful rewrites

Tool	Runtime (seconds)	Memory high-water (kbytes)	Relative program size change
ddisasm	72.81	509215.92	91.90%
e9patch	2.74	105132.97	114.45%
egalito	454.40	10433233.07	169.17%
mctoll	0.00	1405.30	128.22%
multiverse	1195.72	687609.83	870.71%
reopt	169.89	4061695.00	99.61%
retrowrite	114.57	1967393.19	83.78%
revng	703.74	2244106.85	1581.95%
uroboros	19.17	93575.58	148.97%
zipr	233.61	1015891.15	140.05%

the extra runtime dependencies they require that impose significant extra burden when running full test suites in the test environment.

Overall we find a weak correlation with only 258 rewritten programs passing their full test suite of the 395 rewritten programs that passed their Null Function tests. However, it is worth noting that some original binaries (i.e., inputs to the binary rewriter) that pass the Null Function test do not pass the full test suite (e.g., `redis` when compiled with `-Ofast`).

Note that we disable one test in `redis` because it looks for a specific symbol in the stack trace. This is sufficiently *internal* that we believe it does not compromise program soundness for binary rewriters to change this behavior.

4.8 Performance Against Full Test Suite

The performance of rewritten binaries is critical to many use cases for static binary rewriting. If performance degradation exceeds that of dynamic binary rewriting then dynamic rewriting is often a better alternative as it is able to leverage runtime information to more reliably transform program behavior. We report the change in runtime and memory requirements for successfully rewritten programs running against their full test suite in Table 11. Only those rewriters which produced binaries capable of passing all tests are included. With the exception of Reopt-rewritten binaries which had resource consumption at least an order of magnitude over the original, runtime and memory consumption of the rewritten binaries is close to that of the original binary. This is especially true of the memory high-water mark.

5 DISCUSSION

We identify several trends with respect to binary rewriter IR from our results. First, rewriting via LLVM IR appears to be infeasible given the current state of binary type analysis. Only one binary (the trivial `hello-world`) was successfully instrumented for AFL++ using an LLVM rewriter (mctoll). Additionally, non-trivial NOP-transformed binaries successfully produced with reopt had dramatically increased runtime and memory consumption as compared to the original. Second, direct rewriting as performed by Egalito and Zipr successfully produced executables even in the presence of analysis errors; however their output binaries also demonstrated a higher functional failure rate. Conversely, reassemblable disassemblers were more likely to raise errors during re-assembling and re-linking and thus fail to create an executable. Trampoline rewriters such as e9patch are *very* reliable across a wide range of binaries if only additive instrumentation and no modification of existing code is required.

During our evaluation, we communicated with the developers of our evaluated tools to share our partial results and our benchmark set. Unsurprisingly, the best-performing tools in our evaluation, ddisasm and zipr, had sufficient development resources to respond to specific failures encountered in our work. Thus, their performance against this evaluation set likely outperforms their expected performance in general. This is indicative of a *defining* characteristic of binary rewriting at this point in time; binary rewriting is eminently practical in many *particular* cases that have been addressed and considered by tool developers, but impossible in the *general* case as the universe of binary formats and features is simply too large with too many edge cases to handle.

Our evaluation indicates that practical applications of binary rewriting should be preceded by a scoping stage. In this stage, the target binary is classified as either “in scope” or “out of scope” for the binary rewriting tool(s) of interest. While scoping can be accomplished via traditional binary analysis, the high success rate demonstrated by our simple predictive model with trivially collected features shows that accurate scoping can be conducted with little effort. Further, the relative simplicity of our model implies that more reliable and accurate predictive models are likely easily within reach. With such a model, users of binary rewriting tools

Table 9: Comparative runtime of binary rewriting tools

Tool	ddisasm	e9patch	egalito	mctoll	multiverse	reopt	retrowrite	revng	uroboros	zipr
ddisasm	100%	2656.96%	15.95%	35288840.58%	4.40%	42.86%	26.97%	10.35%	689.90%	33.68%
e9patch	3.76%	100%	0.59%	1328163.77%	0.19%	1.61%	2.23%	0.39%	15.49%	1.20%
egalito	626.77%	16963.74%	100%	219758856.52%	18.30%	270.91%	503.69%	64.57%	2123.71%	205.48%
mctoll	0.00%	0.01%	0.00%	100%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
multiverse	2270.67%	51759.60%	546.47%	441563025.00%	100%	749.41%	1983.34%	224.58%	20301.47%	1550.43%
reopt	233.32%	6199.21%	36.91%	82335688.41%	13.34%	100%	166.07%	24.14%	811.10%	74.87%
retrowrite	370.73%	4485.01%	19.85%	32450926.47%	5.04%	60.22%	100%	19.32%	33.45%	50.96%
revng	966.48%	25678.91%	154.87%	341058002.90%	44.53%	414.23%	517.54%	100%	4073.82%	300.14%
uroboros	14.49%	645.59%	4.71%	7081250.00%	0.49%	12.33%	298.95%	2.45%	100%	10.68%
zipr	296.89%	8349.20%	48.67%	104211334.78%	6.45%	133.56%	196.23%	33.32%	936.69%	100%

Table 10: Number of binaries passing their full test suite versus passing the Null Function test

Tool	lighttpd	nginx	redis
original	30/30	60/60	26/30
ddisasm	0/30	60/60	26/30
e9patch	30/30	60/60	26/30
egalito	18/18	18/18	10/10
multiverse	0/0	0/0	0/0
reopt	2/19	4/60	2/8
retrowrite	0/9	16/26	0/0
revng	0/0	0/0	0/0
uroboros	0/0	0/0	0/0
zipr	28/30	58/58	22/30

Table 11: Average performance of rewritten binaries when run against the full test suite

Tool	Runtime (seconds)	Memory High-water Mark (kbytes)
ddisasm	109.43%	100.21%
e9patch	119.53%	99.06%
egalito	104.45%	99.85%
reopt	1324.66%	51937.17%
retrowrite	103.84%	100.17%
zipr	102.50%	102.38%

may quickly ensure their target binaries meet the expectations of the available rewriting tools before initiating expensive binary rewriting tasks. For binaries that are likely to fail during static rewriting, the user could either conserve their resources by forgoing binary rewriting or spend them employing more expensive techniques such as dynamic binary rewriting.

6 CONCLUSION

In this work, we evaluated and compared ten binary rewriting tools on two rewriting tasks across a corpus of 3344 variant binaries produced using three compilers and 34 benchmark programs. Our evaluation measured the performance of the tools themselves as well as the performance and soundness of the rewritten binaries they produce. In general, our evaluation indicates that binary rewriters that lift to high-level machine-independent IRs (e.g., LLVM IR) were much less successful in terms of generality and reliability. Additionally, we identified binary features that are predictive of rewriting success and showed that a simple decision tree model trained on these features can accurately predict whether a particular tool can rewrite a target binary. The findings and artifacts contributed by this evaluation have been made publicly available and are intended to support users and developers of binary rewriting tools and drive rewriter adoption and maturation.

ARTIFACT AVAILABILITY

We have made the full set of artifacts generated in this work including our evaluation infrastructure, corpus of test binaries, predictive models, and the evaluated tools publicly available at:

https://gitlab.com/GrammaTech/lifter-eval [13]

ACKNOWLEDGMENTS

This material is based upon work supported by the Office of Naval Research (ONR) under Contract No. N00014-21-C-1032. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ONR.

REFERENCES

[1] AFL++. 2022. AFL++. https://aflplus.plus/.

[2] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 583–600. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse

[3] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *NDSS*. https://doi.org/10.14722/ndss.2018.23304

[4] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools* (Szeged, Hungary) *(PASTE '11)*. ACM, New York, NY, USA, 9–16. https://doi.org/10.1145/2024569.2024572

[5] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. 2020. Scalable Validation of Binary Lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 655–671. https://doi.org/10.1145/3385412.3385964

[6] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. 131–141.

[7] Artem Dinaburg and Andrew Ruef. 2014. McSema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada.*

[8] Sushant Dinesh. 2019. *RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization.* Ph. D. Dissertation. figshare.

[9] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 151–163.

[10] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. In *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, 1075–1092. https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[11] Inc. Galois. 2021. ReOpt. https://github.com/GaloisInc/reopt.

[12] Google. 2022. Bloaty: a size profiler for binaries. https://github.com/google/bloaty.

[13] Inc. Grammatech. 2022. Lifter Evaluation. https://GitLab.com/GrammaTech/lifter-eval.

[14] Jason D Hiser, Anh Nguyen-Tuong, Michele Co, Benjamin Rodes, Matthew Hall, Clark L Coleman, John C Knight, and Jack W Davidson. 2014. A Framework for Creating Binary Rewriting Tools (Short Paper). In *Dependable Computing Conference (EDCC), 2014 Tenth European.* IEEE, 142–145.

[15] Pantea Kiaei, Cees-Bart Breunesse, Mohsen Ahmadi, Patrick Schaumont, and Jasper van Woudenberg. 2020. Rewrite to Reinforce: Rewriting the Binary to Apply Countermeasures against Fault Injection. *arXiv preprint arXiv:2011.14067* (2020).

[16] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. 2010. PEBIL: Efficient static binary instrumentation for Linux. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS).* 175–183. https://doi.org/10.1109/ISPASS.2010.5452024

[17] Kaiyuan Li, Maverick Woo, and Limin Jia. 2020. On the Generation of Disassembly Ground Truth and the Evaluation of Disassemblers. In *Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation.* 9–14.

[18] Zephyr Software LLC. 2022. IRDB Cookbook Examples. https://git.zephyr-software.com/opensrc/irdb-cookbook-examples.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[20] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbr&#252;cken, Germany) *(ISSTA 2016).* ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/2931037.2931047

[21] Microsoft. 2022. mctoll. https://github.com/microsoft/llvm-mctoll.

[22] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 833–851.

[23] Phoenix. 2021. Phoenix. https://github.com/s3team/phoenix.

[24] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. {RAZOR}: A framework for post-deployment software debloating. In *28th {USENIX} Security Symposium ({USENIX} Security 19).* 1733–1750.

[25] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. 1997. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, Vol. 1997. 1–8.

[26] Eric M. Schulte, Jonathan Dorn, Antonio Flores-Montoya, Aaron Ballman, and Tom Johnson. 2019. GTIRB: Intermediate Representation for Binaries. *ArXiv* abs/1907.02859 (2019).

[27] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001).*

[28] 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. (2016).

[29] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. 2013. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Reverse Engineering (WCRE), 2013 20th Working Conference on.* IEEE, 52–61.

[30] Eli Tilevich and Yannis Smaragdakis. 2005. Binary refactoring: Improving code behind the scenes. In *Proceedings of the 27th international conference on Software engineering.* ACM, 264–273.

[31] Inc. Trail of Bits. 2022. Anvill. https://github.com/lifting-bits/anvill.

[32] Inc. Trail of Bits. 2022. Rellic. https://github.com/lifting-bits/rellic.

[33] Inc. Trail of Bits. 2022. Remill. https://github.com/lifting-bits/remill.

[34] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. 2005. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.* IEEE, 7–12.

[35] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *NDSS.*

[36] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *24th USENIX Security Symposium (USENIX Security 15).* USENIX Association, Washington, D.C., 627–642. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-shuai

[37] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference.* ACM, 299–308.

[38] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. 2019. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)* 52, 3 (2019), 1–37.

[39] David Williams-King. 2022. AFL Setup. https://github.com/columbia/egalito-artefact/blob/master/afl-support/setup.sh#L25.

[40] David Williams-King. 2022. AFL Support. https://github.com/columbia/egalito-artefact/blob/master/afl-support/test-readelf.sh#L10.

[41] David Williams-King. 2022. README. https://github.com/columbia/egalito-artefact/blob/master/README-afl.txt.

[42] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 133–147.

[43] William Woodruff, Niki Carroll, and Sebastiaan Peters. 2021. Differential analysis of x86-64 instruction decoders. In *Proceedings of the Seventh Language-Theoretic Security Workshop (LangSec) at the IEEE Symposium on Security and Privacy.*

[44] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. 2019. {CONFIRM}: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *28th USENIX Security Symposium (USENIX Security 19).* 1805–1821.

[45] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. 2014. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments.* ACM, 129–140.

[46] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries.. In *USENIX Security.* 337–352.

A SUPPLEMENTARY PERFORMANCE TABLES

Table 12: Comparative size of rewritten binaries in the intersection of those programs which are successfully rewritten by both tools. The percentage of the successfully rewritten binary sizes by both tools are calculated as a ratio of the row tool to the column tool. For example, multiverse rewritten binaries are just over 9 times bigger on average than ddisasm rewritten binaries. An entry of "NA" indicates that *no* binaries were successfully rewritten by both tools.

Tool	ddisasm	e9patch	egalito	mctoll	multiverse	reopt	retrowrite	revng	uroboros	zipr
ddisasm	100%	79.65%	51.45%	100.52%	10.67%	92.59%	106.39%	6.47%	64.51%	63.82%
e9patch	125.54%	100%	67.71%	98.09%	13.06%	115.77%	141.65%	7.59%	84.26%	81.72%
egalito	194.36%	147.69%	100%	189.77%	21.58%	167.93%	164.72%	10.54%	95.58%	121.13%
mctoll	99.48%	101.94%	52.69%	100%	27.08%	351.83%	98.89%	3.54%	129.17%	84.60%
multiverse	937.59%	765.59%	463.33%	369.23%	100%	854.06%	907.58%	51.55%	567.15%	626.59%
reopt	108.00%	86.38%	59.55%	28.42%	11.71%	100%	114.93%	5.89%	56.22%	70.62%
retrowrite	93.99%	70.60%	60.71%	101.12%	11.02%	87.01%	100%	6.62%	NA	67.52%
revng	1546.44%	1318.30%	949.20%	2823.92%	193.97%	1696.73%	1509.82%	100%	1463.48%	982.19%
uroboros	155.01%	118.68%	104.63%	77.42%	17.63%	177.89%	NA	6.83%	100%	109.41%
zipr	156.68%	122.37%	82.56%	118.20%	15.96%	141.60%	148.11%	10.18%	91.40%	100%

Table 13: Average size change by section for each rewriting tool

Section	ddisasm	e9patch	egalito	mctoll	multiverse	reopt	retrowrite	revng	uroboros	zipr
.got.plt	100.08%	100.00%	100.67%	89.60%	100.00%	100.29%	100.00%	NA	99.24%	NA
.data	102.94%	100.00%	100.24%	NA	99.99%	103.29%	100.42%	1014.94%	101.87%	NA
.dynamic	97.48%	100.00%	66.53%	97.70%	100.00%	99.82%	101.83%	NA	99.91%	NA
.rela.dyn	87.27%	114.67%	823.91%	54.35%	100.00%	102.65%	97.30%	355.43%	91.19%	NA
.strtab	104.90%	100.00%	77.54%	95.27%	99.26%	76.69%	105.45%	407.41%	485.39%	NA
.dynsym	75.44%	100.00%	101.86%	75.24%	100.00%	110.20%	77.96%	821.27%	99.43%	NA
.dynstr	72.78%	100.00%	101.42%	73.18%	99.99%	104.62%	74.56%	1075.44%	99.75%	NA
.symtab	118.87%	100.00%	84.73%	93.93%	100.00%	107.44%	95.09%	270.77%	1675.22%	NA
.eh_frame_hdr	103.76%	100.00%	NA	110.43%	100.00%	106.36%	93.36%	25.33%	108.05%	NA
.plt	100.05%	100.00%	100.45%	89.60%	100.00%	100.28%	99.81%	990.99%	99.75%	NA
.rela.plt	99.93%	100.00%	99.35%	NA	100.00%	100.30%	99.81%	757.03%	100.00%	NA
.eh_frame	109.75%	100.00%	NA	101.88%	100.00%	125.52%	15.14%	520.42%	111.34%	NA
[ELF Program Headers]	96.37%	100.00%	94.29%	105.53%	144.44%	94.00%	104.79%	NA	97.77%	119.81%
[ELF Section Headers]	97.66%	100.00%	71.70%	94.93%	116.13%	103.62%	92.86%	148.37%	97.57%	98.55%
.rodata	100.10%	100.00%	100.43%	NA	100.00%	99.96%	100.06%	480.80%	100.03%	NA
.text	146.85%	100.00%	161.64%	100.89%	100.00%	240.13%	120.45%	3817.02%	110.17%	NA
[Unmapped]	128.93%	13162.97%	670.47%	350.40%	2285.56%	181.98%	225.89%	22384.87%	165.31%	10.40%

Table 14: Comparative memory high-water mark in kilobytes between rewriting tools. The comparative memory high-water mark across successfully rewritten binaries by both tools is expressed as a percentage of the row tool to the column tool. For example, uroboros' maximum memory consumption is roughly 15.14% of the maximum memory consumption of ddisasm.

Tool	ddisasm	39patch	egalito	mctoll	multiverse	reopt	retrowrite	revng	uroboros	zipr
ddisasm	100%	484.35%	4.69%	36235.17%	76.88%	12.54%	24.70%	22.69%	660.63%	53.68%
e9patch	20.65%	100%	0.99%	7481.13%	13.13%	2.59%	5.39%	4.68%	119.91%	10.57%
egalito	2133.00%	10124.69%	100%	741944.13%	1446.63%	259.70%	499.62%	471.95%	11511.47%	1048.77%
mctoll	0.28%	1.34%	0.01%	100%	0.23%	0.03%	0.09%	0.06%	1.64%	0.14%
multiverse	130.07%	761.64%	6.91%	43096.38%	100%	18.42%	66.16%	38.14%	806.39%	111.25%
reopt	797.64%	3863.39%	38.51%	289025.13%	542.85%	100%	206.36%	180.99%	4336.39%	407.09%
retrowrite	404.93%	1856.78%	20.02%	112051.06%	151.14%	48.46%	100%	94.37%	251.50%	197.87%
revng	440.70%	2134.54%	21.19%	159687.83%	262.22%	55.25%	105.97%	100%	2492.68%	229.02%
uroboros	15.14%	83.40%	0.87%	6083.90%	12.40%	2.31%	39.76%	4.01%	100%	11.28%
zipr	186.28%	945.96%	9.53%	70240.87%	89.89%	24.56%	50.54%	43.66%	886.33%	100%

B PREDICTIVE MODELS

Figure 1: Decision tree to predict the success of AFL instrumentation with ddisasm. Accuracy of 81.47%.

```
def ddisasm_tree(note.abi_tag, interp, strip, rela.plt, pi):
  if not note.abi_tag:
    if not interp:
      if strip:
        if interp:
          return {'FAIL': 50.0, 'PASS': 112.0}
        else: # not interp
          return {'FAIL': 37.0, 'PASS': 33.0}
      else: # not strip
        return {'FAIL': 12.0, 'PASS': 0.0}
    else: # interp
      if rela.plt:
        if interp:
          return {'FAIL': 47.0, 'PASS': 910.0}
        else: # not interp
          return {'FAIL': 92.0, 'PASS': 368.0}
      else: # not rela.plt
        return {'FAIL': 10.0, 'PASS': 0.0}
  else: # note.abi_tag
    if not strip:
      return {'FAIL': 53.0, 'PASS': 0.0}
    else: # strip
      if not interp:
        if interp:
          return {'FAIL': 64.0, 'PASS': 11.0}
        else: # not interp
          return {'FAIL': 22.0, 'PASS': 3.0}
      else: # interp
        if not pi:
          if interp:
            return {'FAIL': 215.0, 'PASS': 168.0}
          else: # not interp
            return {'FAIL': 82.0, 'PASS': 38.0}
        else: # pi
          return {'FAIL': 0.0, 'PASS': 15.0}
```

Figure 2: Decision tree to predict the success of AFL instrumentation with e9patch. Accuracy of 86.06%.

```
def e9patch_tree(pi, note.gnu.build_id, got.plt, interp, strip, note.abi_tag, rela.plt):
  if not pi:
    if note.gnu.build_id:
      return {'FAIL': 723.0, 'PASS': 0.0}
    else: # not note.gnu.build_id
      if got.plt:
        if not note.gnu.build_id:
          if interp:
            return {'FAIL': 3.0, 'PASS': 0.0}
          else: # not interp
            return {'FAIL': 39.0, 'PASS': 6.0}
        else: # note.gnu.build_id
          return {'FAIL': 13.0, 'PASS': 0.0}
      else: # not got.plt
        if not note.gnu.build_id:
          if interp:
            return {'FAIL': 46.0, 'PASS': 0.0}
          else: # not interp
            return {'FAIL': 160.0, 'PASS': 7.0}
        else: # note.gnu.build_id
          return {'FAIL': 58.0, 'PASS': 0.0}
  else: # pi
    if not interp:
      if interp:
        if strip:
          if got.plt:
            if not note.gnu.build_id:
              return {'FAIL': 2.0, 'PASS': 3.0}
            else: # note.gnu.build_id
              return {'FAIL': 9.0, 'PASS': 1.0}
          else: # not got.plt
            if not note.gnu.build_id:
              return {'FAIL': 31.0, 'PASS': 32.0}
            else: # note.gnu.build_id
              return {'FAIL': 22.0, 'PASS': 22.0}
        else: # not strip
          return {'FAIL': 0.0, 'PASS': 15.0}
      else: # not interp
        if note.abi_tag:
          if got.plt:
            return {'FAIL': 23.0, 'PASS': 1.0}
          else: # not got.plt
            if not note.gnu.build_id:
              return {'FAIL': 96.0, 'PASS': 21.0}
            else: # note.gnu.build_id
              return {'FAIL': 6.0, 'PASS': 2.0}
        else: # not note.abi_tag
          return {'FAIL': 53.0, 'PASS': 0.0}
    else: # interp
      if got.plt:
        if not rela.plt:
          return {'FAIL': 12.0, 'PASS': 0.0}
        else: # rela.plt
          if interp:
            return {'FAIL': 17.0, 'PASS': 15.0}
          else: # not interp
            return {'FAIL': 51.0, 'PASS': 48.0}
      else: # not got.plt
        if note.abi_tag:
          if not note.gnu.build_id:
            if interp:
              return {'FAIL': 0.0, 'PASS': 47.0}
            else: # not interp
              return {'FAIL': 80.0, 'PASS': 501.0}
          else: # note.gnu.build_id
            return {'FAIL': 35.0, 'PASS': 132.0}
        else: # not note.abi_tag
          return {'FAIL': 10.0, 'PASS': 0.0}
```

Figure 3: Decision tree to predict the success of AFL instrumentation with mctoll. Accuracy of 98.80%.

```
def mctoll_tree(note.abi_tag, strip, pi, got.plt,
                data.rel.ro, symtab, note.gnu.build_id):
  if note.abi_tag:
    return {'FAIL': 1672.0, 'PASS': 0.0}
  else: # not note.abi_tag
    if strip:
      if pi:
        if not got.plt:
          if data.rel.ro:
            if not symtab:
              return {'FAIL': 5.0, 'PASS': 6.0}
            else: # symtab
              return {'FAIL': 3.0, 'PASS': 0.0}
          else: # not data.rel.ro
            if symtab:
              return {'FAIL': 21.0, 'PASS': 4.0}
            else: # not symtab
              return {'FAIL': 3.0, 'PASS': 0.0}
        else: # got.plt
          return {'FAIL': 17.0, 'PASS': 0.0}
      else: # not pi
        if symtab:
          if got.plt:
            return {'FAIL': 21.0, 'PASS': 0.0}
          else: # not got.plt
            if not note.gnu.build_id:
              return {'FAIL': 98.0, 'PASS': 6.0}
            else: # note.gnu.build_id
              return {'FAIL': 80.0, 'PASS': 3.0}
        else: # not symtab
          return {'FAIL': 69.0, 'PASS': 0.0}
    else: # not strip
      return {'FAIL': 334.0, 'PASS': 0.0}
```

Figure 4: Decision tree to predict the success of AFL instrumentation with retrowrite. Accuracy of 93.02%.

```
def retrowrite_tree(note.gnu.build_id, pi, got.plt,
                    note.abi_tag, rela.plt,
                    data.rel.ro,interp):
  if note.gnu.build_id:
    if not pi:
      return {'FAIL': 531.0, 'PASS': 0.0}
    else: # pi
      if got.plt:
        return {'FAIL': 169.0, 'PASS': 0.0}
      else: # not got.plt
        if note.abi_tag:
          if not note.abi_tag:
            if rela.plt:
              if data.rel.ro:
                return {'FAIL': 36.0, 'PASS': 50.0}
              else: # not data.rel.ro
                return {'FAIL': 78.0, 'PASS': 64.0}
            else: # not rela.plt
              return {'FAIL': 8.0, 'PASS': 0.0}
          else: # note.abi_tag
            if not data.rel.ro:
              return {'FAIL': 64.0, 'PASS': 32.0}
            else: # data.rel.ro
              return {'FAIL': 11.0, 'PASS': 0.0}
        else: # not note.abi_tag
          if interp:
            return {'FAIL': 11.0, 'PASS': 0.0}
          else: # not interp
            if not rela.plt:
              return {'FAIL': 82.0, 'PASS': 36.0}
            else: # rela.plt
              return {'FAIL': 4.0, 'PASS': 0.0}
  else: # not note.gnu.build_id
    return {'FAIL': 1166.0, 'PASS': 0.0}
```

Figure 5: Decision tree to predict the success of AFL instrumentation with zipr. Accuracy of 79.98%.

```
def zipr_tree(got.plt, interp, pi, rela.plt,
              note.gnu.build_id, note.abi_tag,
              strip):
  if not got.plt:
    if got.plt:
      if interp:
        if not interp:
          return {'FAIL': 19.0, 'PASS': 114.0}
        else: # interp
          return {'FAIL': 17.0, 'PASS': 113.0}
      else: # not interp
        if not pi:
          return {'FAIL': 30.0, 'PASS': 103.0}
        else: # pi
          return {'FAIL': 26.0, 'PASS': 108.0}
    else: # not got.plt
      if interp:
        if not pi:
          return {'FAIL': 10.0, 'PASS': 26.0}
        else: # pi
          return {'FAIL': 7.0, 'PASS': 24.0}
      else: # not interp
        if pi:
          if rela.plt:
            return {'FAIL': 14.0, 'PASS': 0.0}
          else: # not rela.plt
            if not note.gnu.build_id:
              if not pi:
                return {'FAIL': 29.0, 'PASS': 5.0}
              else: # pi
                return {'FAIL': 21.0, 'PASS': 10.0}
            else: # note.gnu.build_id
              return {'FAIL': 13.0, 'PASS': 0.0}
        else: # not pi
          return {'FAIL': 0.0, 'PASS': 15.0}
  else: # got.plt
    if not pi:
      if interp:
        if got.plt:
          if not note.abi_tag:
            if not note.gnu.build_id:
              return {'FAIL': 4.0, 'PASS': 10.0}
            else: # note.gnu.build_id
              return {'FAIL': 11.0, 'PASS': 76.0}
          else: # note.abi_tag
            if not note.gnu.build_id:
              return {'FAIL': 0.0, 'PASS': 14.0}
            else: # note.gnu.build_id
              return {'FAIL': 7.0, 'PASS': 29.0}
        else: # not got.plt
          return {'FAIL': 0.0, 'PASS': 16.0}
      else: # not interp
        if not note.abi_tag:
          if not strip:
            if got.plt:
              if not note.gnu.build_id:
                return {'FAIL': 41.0, 'PASS': 133.0}
              else: # note.gnu.build_id
                return {'FAIL': 10.0, 'PASS': 45.0}
            else: # not got.plt
              if not note.gnu.build_id:
                return {'FAIL': 23.0, 'PASS': 43.0}
              else: # note.gnu.build_id
                return {'FAIL': 41.0, 'PASS': 34.0}
          else: # strip
            return {'FAIL': 21.0, 'PASS': 0.0}
        else: # note.abi_tag
          if got.plt:
            if not strip:
              if not note.gnu.build_id:
                return {'FAIL': 63.0, 'PASS': 55.0}
              else: # note.gnu.build_id
                return {'FAIL': 31.0, 'PASS': 27.0}
            else: # strip
              return {'FAIL': 4.0, 'PASS': 0.0}
          else: # not got.plt
            if rela.plt:
              return {'FAIL': 6.0, 'PASS': 0.0}
            else: # not rela.plt
              if not note.gnu.build_id:
                return {'FAIL': 32.0, 'PASS': 9.0}
              else: # note.gnu.build_id
                return {'FAIL': 30.0, 'PASS': 6.0}
    else: # pi
      if not note.gnu.build_id:
        if note.abi_tag:
          return {'FAIL': 30.0, 'PASS': 0.0}
        else: # not note.abi_tag
          if got.plt:
            if not note.abi_tag:
              if interp:
                return {'FAIL': 13.0, 'PASS': 1.0}
              else: # not interp
                return {'FAIL': 134.0, 'PASS': 39.0}
            else: # note.abi_tag
              if interp:
                return {'FAIL': 8.0, 'PASS': 3.0}
              else: # not interp
                return {'FAIL': 98.0, 'PASS': 21.0}
          else: # not got.plt
            if interp:
              return {'FAIL': 0.0, 'PASS': 9.0}
            else: # not interp
              if not note.abi_tag:
                return {'FAIL': 40.0, 'PASS': 22.0}
              else: # note.abi_tag
                return {'FAIL': 37.0, 'PASS': 7.0}
      else: # note.gnu.build_id
        return {'FAIL': 355.0, 'PASS': 0.0}
```