CORNUCOPIA: A Framework for Feedback Guided Generation of Binaries

Vidush Singhal
Purdue University
United States
singhav@purdue.edu

Akul Abhilash Pillai
Purdue University
United States
pillai23@purdue.edu

Charitha Saumya
Purdue University
United States
cgusthin@purdue.edu

Milind Kulkarni
Purdue University
United States
milind@purdue.edu

Aravind Machiry
Purdue University
United States
amachiry@purdue.edu

ABSTRACT

Binary analysis is an important capability required for many security and software engineering applications. Consequently, there are many binary analysis techniques and tools with varied capabilities. However, testing these tools requires a large, varied binary dataset with corresponding source-level information. In this paper, we present CORNUCOPIA, an architecture agnostic automated framework that can generate a plethora of binaries from corresponding program source by exploiting compiler optimizations and feedback-guided learning. Our evaluation shows that CORNUCOPIA was able to generate 309K binaries across four architectures (x86, x64, ARM, MIPS) with an average of 403 binaries for each program and outperforms BINTUNER [53], a similar technique. Our experiments revealed issues with the LLVM optimization scheduler resulting in compiler crashes (~300). Our evaluation of four popular binary analysis tools ANGR, GHIDRA, IDA, and RADARE, using CORNUCOPIA generated binaries, revealed various issues with these tools. Specifically, we found 263 crashes in ANGR and one memory corruption issue in IDA. Our differential testing on the analysis results revealed various semantic bugs in these tools. We also tested machine learning tools, ASM2VEC, SAFE, and DEBIN, that claim to capture binary semantics and show that they perform poorly (*e.g.,* DEBIN F1 score dropped to 12.9% from reported 63.1%) on CORNUCOPIA generated binaries. In summary, our exhaustive evaluation shows that CORNUCOPIA is an effective mechanism to generate binaries for testing binary analysis techniques effectively.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**.

KEYWORDS

Compiler Optimizations, Fuzzing, Automated Binary Generation, Binary Code Difference

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3561152

ACM Reference Format:
Vidush Singhal, Akul Abhilash Pillai, Charitha Saumya, Milind Kulkarni, and Aravind Machiry. 2022. CORNUCOPIA: A Framework for Feedback Guided Generation of Binaries. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3561152

1 INTRODUCTION

Designing proper binary analysis tools is a challenging task. It requires precisely modeling all the units of the underlying Instruction Set Architecture (ISA). Even commonly-used, supposedly-robust tools, such as QEMU, have bugs in precisely modeling certain instructions [28]. One common approach to designing these tools, especially static analysis tools, is to perform incremental development [5]. Specifically, instead of painstakingly modeling all the aspects of the underlying ISA, tool developers model only those instructions and patterns commonly observed in binaries. These common patterns are highly dependent on which binaries developers consider. Without evaluating a representative dataset of binaries, some key patterns may be overlooked, and these tools will thus be less robust. Similarly, Machine Learning (ML) techniques [73] used to solve various binary analysis problems also rely on a varied dataset of binaries for training. For certain security-critical applications such as malware detection [58], a misprediction (*i.e.,* false negative) by the corresponding ML model can be disastrous for the security of the underlying system [35]. In order to mitigate such issues and to build robust ML models, it is important to ensure that the training dataset of binaries is sufficiently varied.

Most existing tools to produce binary datasets [1, 67] use binaries generated using standard optimization flags (*i.e.,* O0, O1, O2, O3, Os, Ofast). Unfortunately, the binaries generated using standard optimization flags frequently miss common idioms [12]. Consequently, analysis tools developed based on these datasets fail to handle certain idioms, resulting in tool failures, as evident from a large number of issues in ANGR [6, 57] and RADARE [15, 52]. These tools enable [49] important security and software maintenance applications such as Control Flow Integrity (CFI) [66], Automated Patching [32], and Binary Rewriting [2, 69]. Failures in these tools impact their usability and delay research progress. Unfortunately, irrespective of an active open source community, academic researchers spend considerable time fixing various robustness issues in these tools [16]. Similarly, ML tools trained using binaries generated from standard

optimization flags (-Ox) are shown to perform poorly on binaries compiled with non-standard optimization flags [53].

We wish to automatically generate well-formed binaries so that binary analysis and ML tools can use the generated datasets to improve their robustness. The generated binaries should have associated high-level structures, specifically source code, to enable the creation of ground-truth information (*i.e.,* through debug symbols) needed by machine learning tools. Existing binary-level techniques [11, 24, 62, 65, 70] use semantics-preserving transformations (*e.g.,* Register Swapping) to generate several semantically-equivalent binaries from a single binary. These techniques are primarily designed for program obfuscation and are based on fixed patterns. Consequently, the number of variants generated for a given binary is limited. Second, these techniques depend on the ability to perform static binary rewriting and reassemblable disassembly, which is known to be a hard problem [69]. Third, as mentioned before, we need to have source code or ground truth information corresponding to the generated binaries. However, generating source code for a given arbitrary binary (*i.e.,* decompilation) is known to be a hard problem [63]. Finally, generating semantics preserving transformations requires a precise model of the underlying ISA, which requires a considerable amount of effort [8, 20]. For instance, even a simple register swapping/renaming transformation, such as renaming register `RCX` to `RDX` in a function, requires knowledge of the ABI. Specifically, we need to know that the function does not use `RCX` or `RDX` for its arguments. To determine this, we need to know the number and type (scalar or not) of parameters [14] for the function and the calling convention used by the function. Both are known to be challenging [26].

Another class of techniques performs semantics-preserving transformations, but at the source level (*e.g.,* tigress [18]) or IR level (*e.g.,* ob-llvm [33]). These techniques focus on ISA-agnostic control flow and data flow related aspects of the program without considering the ISA-dependent instruction sequence or patterns used in the resulting binary. Consequently, these techniques are shown to have *less* or *no* impact on the generated binary [44]. Table 1 shows a summary of the existing techniques along with their drawbacks.

In this paper, we focus on the problem of generating *large numbers of binaries for a given program.* We aim to develop a tool that binary analysis framework developers can easily use to test their framework effectively. Furthermore, We want to have ground truth information (*i.e.,* source code and debug information) for all the generated binaries. We observe that compilers have these precise models of ISA as part of their target code generation component [60]. Most compilers provide various options and target-(in)dependent optimization flags that allow fine-grained control over choices in code generation [54]. Our basic idea is to use these fine-grained optimization flags to generate different binaries. However, for a given program, not all optimization flags affect the program's binary. For instance, the flag `--x86-use-base-pointer` available in `clang` does not affect programs with small local variables. Although individual flags may be ineffective for certain programs, *combinations* of the flags could generate different binaries [12]. For a given program, identifying which flag combinations affect the target binary is a combinatorial problem—intractable, especially when there are a large and growing number of flags (~ 892 usable flags for x86 in `clang-12.0`).

In fact, we tried the brute-force approach of enumerating all the combinations of compiler to compile programs of different sizes. In 12 hours, on average, the brute-force approach was able to generate 197 unique binaries, whereas our approach was able to generate 6,512 (33×) in just 6 hours (half the time).

We present Cornucopia, an automated, architecture-independent framework for generating a plethora of binaries for a given program. Given a source package (*e.g.,* `a2ps.tar.gz`), compiler, and set of all available optimization flags, Cornucopia iteratively learns to produce unique binaries for a given source package by feedback-guided mutation of compiler flags, thus avoiding enumerating all combinations of optimization flags. A recent work, BinTuner [53], also explores the use of compiler flags to generate different variations of binaries for a given program. Although it uses a search-based iterative compilation, BinTuner's goal is not to generate diverse binaries but to generate a binary very different from those generated by general Ox optimization levels. Furthermore, it requires explicit specification of conflicting compilation options in the form of first-order formulas, which must be specified for every ISA and compiler combination. This requires an in-depth understanding of various compiler options, which involves considerable effort and conflicts with our requirement for an easy-to-use tool. Finally, as shown in Section 4.3, BinTuner's fitness function is inferior to Cornucopia for generating a plethora of diverse binaries. The latter generated 8X more binaries than BinTuner in a given time.

Our evaluation shows that Cornucopia, in 6 hours, can generate, on average, 403 binaries per program across all architectures. In addition, standard tools for evaluating binary differences show that these binary variants are highly varied (refer our extended report [59]) .

Generating a large number of binary variants is only useful if those variants expose interesting behaviors in the software toolchain. The binaries generated by Cornucopia revealed various issues in current static analysis and ML tools, showing the inadequacy of the current methods to make these tools robust. This shows that Cornucopia generates binaries that can be used to improve the robustness of binary analysis tools. Additionally, we observed that Cornucopia can also be used to test the optimization scheduler in compilers to find issues related to optimization dependencies [61]. We found issues with the LLVM optimization scheduler which resulted in compiler crashes ~300. In summary, the following are our contributions:

- We present Cornucopia, a feedback-guided mutation technique to efficiently find sets of compiler optimization flags that produce different binaries for a given application and show that it outperforms BinTuner (Section 4.3), a recent approach that tries to find optimization flags resulting in a large binary code difference.
- Our evaluation shows that Cornucopia generates a large number of unique binaries for a given program, and these binaries differ significantly from those generated using standard optimization levels (Section 4.3.2).
- Our evaluation of existing binary analysis tools and machine learning tools with Cornucopia generated binaries revealed various robustness issues (*i.e.,* 263 crashes in angr and one

Technique	Binaries Have corresponding source code?	ISA Specification Not Needed?	Reassemblable Disassembly Not Needed?	Affects Generated Binaries?	Generates Large Number of Binaries?
Compilation Through Standard Optimization levels [1, 63, 72] (e.g., O0, O1, O3, etc)	✓	✓	✓	✓	✗
Source Level Transformations [20, 69] (e.g., CFG flattening [50])	✓	✓	✓	❍	❍
IR level Transformations [36, 66] (e.g., Deadcode Insertion [12])	✗	✓	✓	❍	❍
Binary Level Transformations [13, 27, 65, 70, 75] (e.g., Register renaming [25])	✗	✗	✗	✓	✓
CORNUCOPIA (Our System)	✓	✓	✓	✓	✓

Table 1: *Comparison of CORNUCOPIA to other binary generation techniques.* For each of the feature, we indicate whether the technique fully supports (✓), partially supports (❍), or does not support (✗) it.

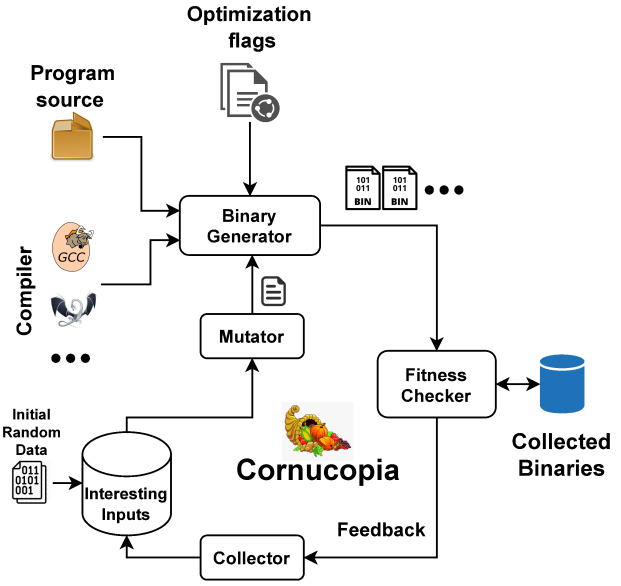

Figure 1: Overview of CORNUCOPIA.

in IDA), semantic issues (Section 4.4), and model performance issues (Section 4.5), demonstrating the utility of CORNUCOPIA in testing existing tools.

- We show that CORNUCOPIA is also effective at testing optimization schedulers in compilers by finding issues with the LLVM optimization scheduler which resulted in ~300 compiler crashes .
- The source code and generated binaries of CORNUCOPIA is available at https://doi.org/10.5281/zenodo.7039858. Refer to our website for more details: https://binarygeneration.github.io/

2 OVERVIEW

This section presents an overview of CORNUCOPIA, as shown in Figure 1. The core technique of CORNUCOPIA is the identification of the set of compiler flags that affect the binary generated from the given source. CORNUCOPIA starts with the program source package S, a compiler C, the list of all flags O supported by the compiler, and an initial $|O|$ (*i.e.,* total number of flags) bytes of random data, used as an initial input.

CORNUCOPIA uses feedback-guided mutation to select compiler flags that have a high probability of changing the structure of the binary, as determined by a configurable fitness function. The *mutator* takes one of the interesting inputs (initially random data), mutates it, and sends it to the *binary generator*. The binary generator uses the data to select a certain subset of compiler flags $o_i \subseteq O$. (In other words, the input is used as a seed to select which compiler flags are used.) These selected compiler flags are used to compile S with C to get a set of binaries b. Note that each source package can result in multiple binaries. For instance, compiling `binutils.tar.gz` package results in 19 binaries, such as `objdump`, `nm`, etc. The generated binaries B are sent to a *fitness checker*, which checks if these binaries are different from previously seen binaries and stores the newly-seen binaries, $B_{new} \subseteq B$, into a database. CORNUCOPIA's fitness checker measures how *different* the binaries in B_{new} are from all the previously seen binaries from the same source package. The measure of difference is converted into a floating-point number and sent as feedback to our collector. The *collector* checks if the feedback value is greater than 0. If yes, it saves the corresponding input (generated by the mutator) into a weighted list of *interesting inputs* (inputs that yield differing binaries).

In the next iteration, the mutator again picks an input from the list of interesting inputs, such that the probability of picking an input is proportional to its feedback value. (This weighted sample means that inputs corresponding to compiler flags that created more varied binaries are preferred.) The selected input is mutated and sent to the binary generator, and the process continues. All the generated binaries will be saved into the database, and similar to the random testing process, the user can stop CORNUCOPIA when she is satisfied with the generated binaries.

3 DESIGN

The design of CORNUCOPIA is built around the way fuzzing frameworks work, which expect to execute an "input" on a "program", generate an output, and from that output use a fitness function to decide whether or how to perturb the input. Crucially, in our setting, the "program" we are fuzzing is the combination of a compiler plus

a program to be compiled (*e.g.,* LLVM plus `objdump`). The *input* is the *compiler flags*. Section 3.1 describes how an input is mapped to compiler flags, and thence to generating an output. Section 3.2 describes how we design our fitness function.

3.1 Binary Generator

The binary generator maps input bytes to compilation flags and uses these flags to compile a given source package to get a set of binaries.

Mapping bytes to compiler flags: For most of the flags, we map each input byte to a compiler flag. The corresponding byte value indicates whether the option is selected or not. However, directly using the byte value will result in unnecessary bias. For instance, consider that we enable an option by just checking whether the value of the byte is greater than 0. There is a 99% (or 255/256) chance that the option is enabled, whereas there is only a 1% (or 1/256) chance that the option will be disabled. To avoid this bias, we use a modulus operation. Specifically, we compute `byte_value mod 2` and enable the flag if the resulting value is 1. Similarly, for flags that expect a value from a fixed list, we use modulus to select a value uniformly from that list. For instance, for `--frame-pointer=<value>`, the `<value>` can be either `all`, `non-leaf`, or `none`. We use `byte_value mod 4` and enable the flag if the resulting value is greater than 0 and the `<value>` can be either `all`, `non-leaf`, or `none` depending on whether the modulus result is 1 2 or 3 respectively.

For flags that take raw integers, we use 2 bytes, where the first byte (`mod 2`) indicates whether the option is enabled, and if enabled, the second byte is the value for the flag. For instance, we map 2 bytes to the flag `--stack-alignment=<uint>`. The flag is selected when the `first_byte_value mod 2` is 1 and the second byte is passed for `<uint>`, *i.e.,* `--stack-alignment=<second_byte_value>`.

We will ignore additional bytes if the input has more bytes than all the compiler flags. Similarly, we will not select the corresponding flags if the input has fewer bytes.

Compiling using the selected flags: We use a dynamic approach by hooking into the build process and dynamically modifying every compiler invocation to include only the selected flags. For instance, consider that our target compiler is `clang` and selected options are `--addrsig` and `--tailcallopt`. Our dynamic hook will replace every compiler invocation, say `gcc -O2 <source file(s)>`, with `clang --addrsig --tailcallopt <source file(s)>`. We also include all the preprocessor directives (*e.g.,* `-D..`) and linker flags that were part of the original compiler invocation.

Handling conflicting flags: The compiler flags can have constraints, including adverse interactions and dependency relationships. Few flags can negatively influence each other, and turning them on together leads to a compilation error. Some other flags may only work when another flag is specified. For example, `-ftree-slp-vectorize` may not have an effect when loop unrolling is disabled because SLP vectorizer may not have opportunities to vectorize the loop body if the loop is not unrolled.

Automatically identifying conflicting compiler flags is a combinatorial problem *i.e.,* requires enumerating all the possible flag combinations, which is intractable when there are a large number of growing flags (~ 892 for x86 in `clang-12.0`). On the other hand, manually specifying conflicting flags for each compiler, as in BinTuner, requires considerable effort. We use a feedback-driven approach to handle this. Specifically, if the compilation fails with selected flags, we compile using a default set of predefined flags (*e.g.,* `-O0`) and generate corresponding binaries. Since the selection of any conflicting flags results in the same binary (*i.e.,* the one built with default flags), the fitness function (Section 3.2) will return a score of *zero* for these binaries. The *zero* score will cause the corresponding input to be discarded by our collector, thereby steering Cornucopia away from generating inputs that result in conflicting compiler flags.

3.2 Fitness Checker

The goal of the fitness checker is to compute how different the provided binaries are from all the previously generated binaries from the same source package. We call this result Difference Score (DScore).

The score computation mechanism should be efficient. Otherwise, it will become a performance bottleneck, and then the overall cost will increase drastically. Existing binary diffing techniques, such as BinDiff [22], require disassembling the binary and performing lightweight analysis, increasing their execution time. For instance, BinDiff takes ~5 min for a medium-sized binary.

There are well-known techniques in malware signature research that use heuristic methods to compute the similarity between two binaries. We explore two such techniques and propose a custom difference score based on the percentage of unique functions. In all of these techniques, the computed DScore is a floating-point number ranging from 0.0 to 1.0, where a larger value indicates a bigger difference. As an optimization, before computing DScore, we check if the binary is not unique *i.e.,* if we have already seen the exact binary, then we immediately return 0. If the provided binaries are unique *i.e., DScore is greater than 0*, the fitness checker also stores these binaries in a database. We explore the following techniques to compute the DScore of a given binary.

Piecewise Hashing: Piecewise hashing or fuzzy hashing [36] is a well-known technique to compare binaries. The comparison of fuzzy hashes results in a value ranging from 0.0 to 1.0 (the higher, the more different). We explore two approaches to compute the DScore based on the piecewise hashing.

Piecewise average (P_a): Here, we compute the difference in the piecewise hash of the given binary with all the previously seen binaries. The final DScore is the average of all the hash difference scores. The intuition behind the average is to compute a score that captures how different the current binary is when compared to *all* the previously seen binaries.

Piecewise minimum (P_m): This technique is similar to the average one above. However, we select the minimum hash difference value instead of the average as the final DScore. The intuition behind the minimum is to prioritize the generation of binaries that differ *largely* from all the previously seen binaries. If we consider the binary generation as a graph traversal, the average strategy can be considered as a Breadth-First traversal, whereas the minimum strategy is a Depth-First traversal. We *do not* consider the maximum value because it unnecessarily prioritizes generating the same kind of binaries. But, the goal of Cornucopia is to maximize the generation of different binaries. For instance, consider a new binary b with piecewise hash similarity of 0.9, 0.1, and 0.4 against binaries x, y and, z, respectively. Using the maximum value would

return 0.9 as the DScore, thus maximizing the generation of binaries similar to b. However, the hash difference value 0.1 indicates that binary b is very similar to y. Hence, using the maximum value may unnecessarily prioritize the generation of similar binaries and decrease the overall variety of binaries.

Normalized Compression Distance: Normalized Compression Distance (NCD) is another well-known technique to compute difference based on an information-theoretic measure [3]. Specifically, NCD infers the degree of similarity between arbitrary byte sequences by the amount of space saved after compression. Previous works [53] which use NCD have shown to be effective at capturing the difference between two arbitrary byte sequences. NCD score ranges from 0.0 to 1.0 (the higher, the more different). Similar to Piecewise hashing (Section 3.2), we define **NCD average (N_a)** and **NCD minimum (N_m)**.

Percentage of Unique Functions (F_h): Here, we compute the difference score as the percentage of unique functions in the provided binary. We determine unique functions as follows: For each function, we compute function hash, which is the hash of the binary code of the function. We use this function hash to see if any previously seen binaries have a function with the same hash. If not, the function is considered unique. Finally, the DScore is computed as the percentage of unique functions over the total number of functions in the binary. The intuition here is to use function level similarity rather than byte-sequences based similarity techniques as used in the previous two approaches.

3.3 Collector and Mutator

The collector receives the feedback (*i.e.,* DScore) for each input and stores the input in a weighted list according to the value of the score. The collector discards inputs with a feedback score of 0. The weighted list is organized such that the probability of selecting an element from the list is proportional to its feedback score.

The mutator selects one or more inputs from the weighted list and performs various mutations on the bytes of the inputs. We use mutation strategies, such as *bit flips*, *byte flips*, and *splicing*, that are shown to be effective in fuzz testing [40].

Refer our extended report [59] for implementation details.

4 EVALUATION

We evaluate Cornucopia to demonstrate its effectiveness in generating binaries and their ability to test the robustness of various binary analysis tools. We pose the following research questions to guide our evaluation:

RQ1: Effectiveness: How effective is Cornucopia in generating binaries, and how do different fitness metrics affect the quality and quantity of the generated binaries?

RQ2: Cornucopia vs. BinTuner: How effective is Cornucopia compared to BinTuner, a recent approach that also uses compiler flags to generate binaries?

RQ3: Applicability to test static analysis tools: How effective is the dataset generated by Cornucopia in testing binary static analysis tools?

RQ4: Applicability to test ML tools: How effective is the dataset generated by Cornucopia in tesing ML tools?

4.1 Setup

4.1.1 Dataset and Compiler. We choose `clang` (or LLVM) version 12 as our target compiler, which is the latest and most stable version available during our experimentation. Our binary generator for `clang` uses pre-generated LLVM Bitcode files as an optimization to avoid rerunning frontend for the same sources.

We collected source packages by scrapping official Debian package repositories, compiled them, and randomly selected 191 bitcode files for each of the four popular architectures,*i.e.,* x86, x64, ARM, and MIPS. We will refer to individual binaries or bitcode files as programs. Table 2 shows the number of programs selected and available optimization flags in `clang` for each architecture. Note that the number of programs is limited by resource constraints; specifically, the availability of machines at our disposal.

4.1.2 Machine Setup and Runtime. We used a server with Intel Xeon 5215 CPU and ran Cornucopia on each program for 6 hours. We ensured that each program ran on a processor core and avoided overloading the server.

4.2 Effectiveness

As explained in Section 3.2, there are various lightweight approaches to compute the difference score that can guide our mutations. There is also another approach, as suggested by a recent work [53] where they take the NCD score of the binary with the binary compiled with `-O0` as the difference score, which we denote as N_o. First, we will evaluate the relative effectiveness of our approaches P_a, P_m, N_a, N_m, and, F_h along with N_o.

4.2.1 Effectiveness of different computation approaches. We choose three programs of different sizes `eot2ttf` (5.5K, small), `lscpu` (270K, medium), and, `nab_r` (1.1M, large) for this experiment. For each of these programs, we ran Cornucopia with different approaches for six hours each. In summary, we had 18 (6 approaches * 3 programs) variations, with each running for six hours. To normalize the effects of randomness, we repeated the whole experiment eight times. We found that our function hash mechanism (F_h) resulted in the largest number of unique binaries generated for all three programs for most of the iterations. The second best technique is Fuzzy Hashing (PIECEWISE) minimum (P_m).

To compare the quality of the generated binaries, we computed the NCD score of each binary against the non-optimized *i.e.,* `-O0` compiled) binary. We found that, on average, the binaries generated by F_h have the highest difference score (*i.e., more different variants*) of ***0.79*** compared to all the other fitness functions.

This shows that our F_h technique to compute difference score is both quantitatively (i.e., more unique binaries in a fixed interval of time) and qualitatively (i.e., more different binaries) more effective at generating unique binaries when used with Cornucopia.

There are two main reasons for the improved effectiveness of F_h: (i) Most of the optimizations in compilers are intraprocedural and work independently on each function. (ii) Functions within a program share similar characteristics [46, 48]. For instance, most of the functions in a string processing library work on strings *i.e.,* `char *` type variables. Hence optimization flags that affect a function in a program most likely also affect other functions in the same program as these functions share similar characteristics. Our F_h approach

exploits this by assigning a higher score to the flag combinations that affect more functions in the program.

We also ran the experiment by avoiding the precise difference score but rather using a 1/0 binary feedback, *i.e.,* whether the generated binary is different (1) or not (0). We observed that all approaches suffered and generated fewer binaries compared to the precise difference score versions. This indicates that using a precise difference score is important for generating large number of unique binaries. The potential reason is that using a precise score helps in guiding the search towards more productive flag combinations while 1/0 will do a random search.

4.2.2 Binary Generation Effectiveness. We use the most effective difference score approach,*i.e.,* function hash (F_h), to evaluate the overall effectiveness of Cornucopia. As mentioned in Section 4.1.2, we ran Cornucopia for 6 hours for each program-architecture combination. The summary of the results is shown in Table 2. In total Cornucopia generated 308,269 unique variants across four architectures for 191 programs, with an average of 403 and median of 413 variants per program across all the architectures (The fine-grained split is discussed in our extended report [59]).

Variants across each architecture: Interestingly, as shown in Table 2 the number of generated binaries differs across architectures. Specifically, there are ~15% more binaries in ARM and MIPS, which have a Reduced Instruction Set Computer (RISC) ISA, compared to x86 and x64, having a Complex Instruction Set Computer (CISC) ISA.

The main reason for this is the difference in the underlying ISA and corresponding optimization opportunities. There are more general-purpose registers in ARM and MIPS than x86 and x64, which increases the compiler's choices for register allocation. An example illustration is in one of our binaries as shown in Figure 2, here compiler choose `r12` and `r3` in the left version v/s `r3` and `r4` in the right version, this further caused register spill (line 7 and 17) to occur in the right version. Furthermore, the fixed-length instructions in ARM and MIPS results in relatively dense basic blocks, *i.e.,* the average number of *instructions* in a basic block are more than in x86 and x64 [13]. This further increases optimization opportunities.

We evaluated Cornucopia on other aspects and presented the results in our extended report [59]. Our results show that Cornucopia is effective at generating a large number of different binaries and can explore the variants that are not covered by the standard optimization levels *i.e.,* O0, O1, O2, and O3.

4.2.3 Compiler Crashes. Although unintended, Cornucopia could be used to test optimization schedulers in compilers. As explained in Section 3.1, our binary generator repeatedly invokes the compiler with different combinations of optimization flags on various programs. Consequently, while generating binaries for different programs, Cornucopia is essentially testing optimization schedulers, although in a blackbox manner. Nonetheless, in our experiments, *we found approx. 300 crashes (i.e., `segfaults`) in the optimization scheduler of* `clang`. An example of one such crash is shown in our extended report [59] . We analyzed one of these crashes and identified that the `--pre-RA-sched=vliw-td` optimization flag is the root cause. This is not a trivial issue to find because triggering the crash requires specific program structure. We reported all our crashes and have been acknowledged by the LLVM team as real bugs. They are currently working on fixing these bugs.

We also extended Cornucopia with `gcc` and presented its results in our extended report [59].

Arch (Available Flags)	Binaries	Avg. Binaries Per Program
x86 (892)	63,197	330.87
x64 (892)	74,169	388.32
ARM (876)	83,701	438.23
MIPS (866)	87,192	456.50
Grand Total	308,269	N/A

Table 2: Performance of Cornucopia: The number of binaries generated for each architecture for 191 programs. Each program-architecture combination is run for 6 hours. The number in the parenthesis show the total number of available optimization flags for that architecture.

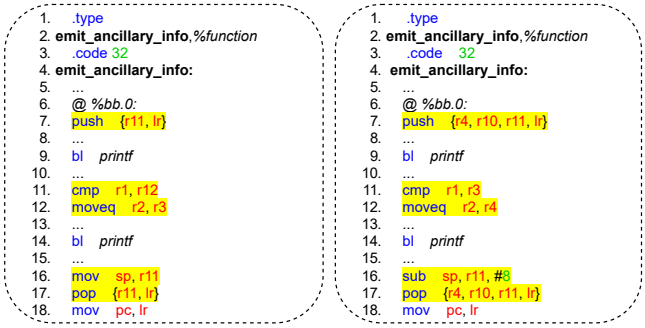

Figure 2: Figure showing ARM assembly for 2 different variants generated from source program `cat`. Variant A uses registers (r12, r3) in place of (r3, r4) (Variant B) for the same function.

4.3 Cornucopia vs. BinTuner

As mentioned in Section 1, BinTuner uses a search-based iterative compilation (based on OpenTuner [7]) to find optimization sequences that can maximize the amount of binary code differences. BinTuner requires an explicit specification of conflicting compiler flags in the form of first-order logic formulas, which requires an in-depth understanding of the flags. This process can be tedious, especially when we need to do this for every architecture supported by the compiler (*i.e.,* x86, x64, ARM, MIPS, etc) and for all desired compiler versions. This imposes considerable *overhead for binary analysis tool developers to use BinTuner.* Furthermore, the implementation of *BinTuner does not support parallelism*, and as such, BinTuner cannot be used in a multi-processor/multi-threaded manner to improve its throughput.

However, Cornucopia only requires specifying the compiler and a corresponding list of supported optimization flags. It does not require *specifying conflicting flags.* Our feedback-driven mechanism (Section 3.1) enables Cornucopia to automatically steer away from

using conflicting flags. The modular design of Cornucopia enables it to be trivially parallelizable by using multiple mutators, all sharing the same interesting inputs source. As shown in Section 4.2, running Cornucopia in parallel mode with six instances resulted in an average of 21X more binaries.

To have an analytical comparison, we perform the following two experiments on the programs on which BinTuner was evaluated. Specifically, we use SPECint 2006, Coreutils, and OpenSSL.

4.3.1 Cornucopia with BinTuner's fitness function (C_b). In this first experiment, we evaluate the binary generation effectiveness of BinTuner's fitness function when used in Cornucopia. Specifically, as in BinTuner, we use NCD score of the generated binary with its `-O0` version as the feedback (*i.e.,* DScore) for the collector in Cornucopia, denoted as C_b.

On average C_b generated 52 binaries vs 450 generated by Cornucopia with the function hash score (F_h). The Figure 7 shows the results across all the programs (Note that the y-axis is in logarithmic (base 10) scale). Except for `447.dealII` and `483.xalancbmk`, Cornucopia generated a large number of binaries, specifically, ~7X more than C_b. The low yield in `447.dealII` and `483.xalancbmk` is because of their large size and the randomness in mutation techniques having less time to explore other effective optimization flag combinations. The reason for the increased effectivenss of Cornucopia is because BinTuner's fitness function (NCD with `-O0`) maximizes the generation of a highly different binary rather than generating a large number of diverse binaries. For instance, C_b likely will not generate highly different binaries that have the same NCD score with `-O0`.

4.3.2 Binary Generation Effectiveness. For this experiment, we ran Cornucopia for 6 hours and BinTuner until it converges or 6 hours (whichever is the latest). On average BinTuner generated 48 binaries vs 450 generated by Cornucopia with the function hash score (F_h), with Figure 7 showing the results across all the programs. Except for five programs, Cornucopia was able to generate more binaries (~8X on average) than BinTuner. The low yield for a few programs is because of their large size and Cornucopia getting less number of iterations in identifying the optimization flags that are effective for these binaries. However, BinTuner, based on OpenTuner [7], uses more systematic exploratory techniques and can quickly identify the potent optimization flags. For instance, the bitcode file for `483.xalancbmk` is 13MB in size, and compilation of it takes ~ 6 minutes. Consequently, Cornucopia gets less time to explore different flag combinations and learn which flags are effective. We confirmed this by running Cornucopia in parallel mode with six cores and observed that we got considerably more binaries than BinTuner.

4.3.3 Quality of the Generated Binaries. We used BinDiff scores to evaluate the quality of binaries generated by different techniques (BinTuner, C_b, and Cornucopia) and Figure 6 shows the cumulative distributive function (CDF) of the scores across all binaries generated for all programs by each of the corresponding techniques. [1] The score ranges from 0 to 1, and it indicates the amount of difference (*i.e.,* larger the score higher the difference). First, as expected, BinTuner was able to generate binaries with the largest difference (~0.95) against its `-O0` and `-O3` versions. However, its steeper curve shows little variance, i.e., most of the BinTuner generated binaries are similar and have high diffence against its `-O0` and `-O3` versions. The less steep curves of Cornucopia and C_b show that they were able to generate more varied binaries, albeit with a lower difference (~0.45) against its `-O0` and `-O3` versions.

We also compared the best binary (*i.e.,* with the highest BinDiff score) generated by BinTuner with the binaries generated by Cornucopia. The Figure 5 shows the CDF of the corresponding score. The steeper curve towards the right indicates that *most of the Cornucopia generated binaries are quite different from those of BinTuner's.* Specifically, 50% of the binaries have their BinDiff scores between 0.75-0.95. This shows that Cornucopia is exploring the binary generation space different from that of BinTuner. In summary, BinTuner is effective at generating binaries highly different from its `-O0`/`-O3` version, but the generated binaries have less variance. However, Cornucopia is a complementary approach and can efficiently generate a large number of binaries with relatively high variance by exploring different binary generation spaces.

4.4 Applicability to Test Static Analysis Tools

We used four popular binary static analysis tools, *i.e.,* Free and open source: angr, Ghidra, and radare; Commercial: ida to evaluate the effectiveness of Cornucopia generated binaries in testing these tools. We choose analyses that are supported by all these tools. Specifically, we choose the following:

Function Boundary Detection (FBD) [9]: This analysis generates a set of function boundaries, where each boundary is a pair of addresses indicating the address of the first and last instruction of a function. We got the ground truth information for FBD from debug information [23] of binaries, specifically, the symbol table [75].

Calling Convention Recovery (CCR): This analysis aims to find the signature [42] of all functions in the binary. For our experiment, we only consider the *number* of parameters. Like FBD, we got the CCR ground truth for each binary using the debug information embedded in it.

To test these two analyses, we compare the ground truth of each binary with the results produced by each tool. For each analysis, we assigned a fixed time of 24 hours for each architecture, randomly picked binaries, and tested them with each tool with a timeout of 10 minutes - most of the tools were able to complete within the timeout except for angr, which timed out for a relatively few large binaries.

Table 3 shows the result across the selected tools. Here, F_s indicate the number of binaries with single tool failures, *i.e.,* only the corresponding tool failed. F_m indicate multi-tool failures, *i.e.,* two or more tools failed. Finally S_a indicates binaries where all tools succeeded, *i.e.,* all tools correctly identified function boundaries for these binaries.

For FBD (Top part of Table 3), on average, all tools correctly identified boundaries for only 19.97% of the binaries across all architectures. Unfortunately, *none* of the tools correctly identified function boundaries for 42.15% of the binaries as indicated by the last row of F_m column. For instance, for a binary of `fallocate` compiled for MIPS, with 172 functions, all the tools except Ghidra failed to

[1] A point (x, y) on a line indicates y% of the binaries have their BinDiff score less than or equal to x.

in Figure 8, we find that radare fails to detect the blocks after the address `0x14dc4` (left side). In comparison, angr CFG (right side) accurately detects the blocks after this address.

We have dissected the results further in Table 4. We categorized the divergence of each tool based on the underlying root causes, *i.e.,* Mismatch in the number of basic blocks (**N**); Number of basic block matches, but the starting addresses differ (**A**) or the ending address or the size of one or more basic blocks differ (**S**), or the edges do not match (**E**); Incomplete output (*i.e.,* tool had an internal failure and did not return any basic blocks) (**P**) and finally timeout (**T**).

Here we find that most tools diverge on the number of basic blocks for a given function, except for radare, in which case most divergences are due to incomplete output, *i.e.,* tool failures. The **N** divergences in ida for x64 are mostly due to failures in tail-call detection, particularly the reason for almost 50% of these seem to be stray `ud2` instructions after a tail call. Although the number of **N** divergences looks significant for angr, further analysis revealed that approximately 99% of these failures are cases where angr chooses to merge jumps to the immediate next address with no other edges into the same basic block. Although this deviates from the approach the other tools take, it can be considered a design choice. Nonetheless, all tools also have internal failures while computing CFG, as indicated by the **P** column - these cases represent bugs in the underlying tools and can assist developers in fixing the underlying issues. We are in the process of organizing these results with appropriate reproducer scripts and reporting them to the corresponding tool developers.

Summary: The analyzed tools have been previously tested with binaries generated using standard optimization levels [67]. Our results indicate that Cornucopia generates binaries that can effectively reveal issues (missed by regular binaries) in static analysis tools. Consequently, Cornucopia can be used to supplement the existing binary datasets to test and further improve binary static analysis tools.

Impact: Our results also raise interesting questions about evaluating advanced binary analysis techniques based on the above tools. For instance, Consider OSPREY [77], a recent type inference technique on binaries based on BDA [78] which uses radare for disassembly and CFG. Our evaluation shows issues with radare for function boundary detection and CFG recovery. However, OSPREY ignores functions that are missed by radare and perform comparative evaluation on ida and Ghidra and show that OSPREY performs better on those functions detected by all these. However, the functions detected by all these tools, which, as we show in our evaluation (Section 4.4) is considerably less. This raises questions about the actual effectiveness of OSPREY as ida and Ghidra may be better or worse on functions missed by radare. This problem becomes severe when we compare similar techniques built using different binary analysis tools. We strongly suggest that binary analysis research should pay particular attention to comparative evaluation, especially when using different binary analysis tools.

Tool Crashes. We also found that angr and ida *crashed* on certain binaries. *Specifically,* angr *crashed on 263 binaries and* ida *on one binary.* For angr, the crashes are in their python framework, whereas for ida, the crash is in the `libdwarf` library. All our issues have been reported and acknowledged by corresponding developers. These issues are being actively fixed.

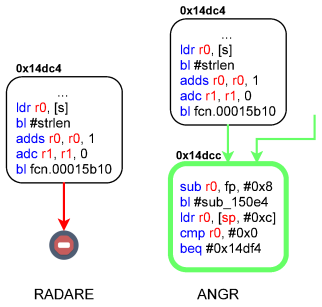

Figure 8: CFGs that show the difference in basic blocks between radare and angr. For radare, there are no basic blocks after the address `0x14dc4`. Whereas for angr there are basic blocks after `0x14dc4`.

4.5 Applicability to Test ML tools

In this section, we will explore the second application of testing the robustness of ML tools on the binaries generated by Cornucopia. We selected the following two recent tools, as these are open-source and claim to have high accuracy.

Binary diffing techniques (Asm2Vec [21] and SAFE [45]): These are representation learning techniques based on neural networks. They propose a representation of binaries into a vector space such that binaries will be close. In other words, the distance between the vector representations of two should be minimal, ideally 0. As in these papers, we use cosine similarity to measure the difference between the generated vectors. Specifically, we compute *Inverse cosine similarity* (*i.e.,* 1 - cosine similarity) denoted as CS_I; a large value of CS_I indicates a higher difference. Ideally, we would want the CS_I to be very low for all the binaries for the same program. However, this is not the case. We got the pre-trained models for these two tools and used the corresponding vectors to compute the CS_I of the generated binaries. Our results as shown that *Cornucopia was able to generate binaries with higher CS_I scores than O3 for all the programs.* A detailed analysis of results is shown in our extended report [59] . This shows that Cornucopia can generate binaries that cannot be detected as similar by the existing techniques. We suggest that these techniques should use Cornucopia to improve their dataset, which could help in building more accurate models.

Debug information prediction (Debin [29]): This technique combines two complementary probabilistic models to predict *types of variables in a stripped binary.* Their evaluation shows that on average, Debin has an F1 score of 67%. We used their pre-trained model and tested its accuracy on each of the binaries generated by Cornucopia. The Table 5 shows the results of our experiment. Although Debin uses binaries of different optimization levels to train their model, it still performs extremely poorly on the binaries generated by Cornucopia, with F1 score dropping to 12.9% (x86), 18.2% (x64) and, 13.6% (ARM) from the reported 67%. We tried to use StateFormer [50], a recent learning-based tool to predict types. However, the pre-built model and the dataset are inaccessible and did not receive help from the authors as well. Nonetheless, our results on other ML techniques show that existing approaches to generate binary datasets are inadequate and Cornucopia can help

Arch.	Randomly Sampled Binaries	Function Boundary Detection					
		F_s				F_m	S_a
		ANGR	GHIDRA	IDA	RADARE		
x86	4,600	**271 (5.89%)**	492 (10.7%)	1,019 (22.15%)	**4,588 (99.74%)**	1,160 (25.22%)	12 (0.26%)
x64	3,516	**213 (6.06%)**	302 (8.59%)	255 (7.25%)	**2,349 (66.81%)**	563 (16.01%)	1,092 (31.06%)
ARM	5,382	**2,388 (44.37%)**	2,891 (53.72%)	2,872 (53.36%)	**3,324 (61.76%)**	3,020 (56.11%)	1,485 (27.59%)
MIPS	4,818	2,744 (56.95%)	2,149 (44.6%)	**679 (14.09%)**	**3,750 (77.83%)**	2,977 (61.79%)	1,068 (22.17%)
Total	18,316	5,616 (30.66%)	5,834 (31.85%)	**4,825 (26.34%)**	**14,011 (76.50%)**	7,720 (42.15%)	3,657 (19.97%)

Arch.	Total No. of Functions*	Calling Convention Recovery					
		F_s				F_m	S_a
		ANGR	GHIDRA	IDA	RADARE		
x86	49,546	**85 (0.17%)**	299 (0.60%)	740 (1.49%)	**9,162 (18.49%)**	18,295 (36.92%)	20,965 (42.31%)
x64	80,174	1,732 (2.16%)	**470 (0.58%)**	4,521 (5.64%)	**16,018 (19.98%)**	17,695 (22.07%)	39,738 (49.56%)
ARM	228,107	13,906 (6.10%)	**816 (0.36%)**	5,012 (2.20%)	**34,892 (15.30%)**	141,680 (62.11%)	31,801 (13.94%)
MIPS	132,461	390 (0.29%)	**129 (0.10%)**	356 (0.27%)	**57,729 (43.58%)**	66,028 (49.85%)	7,829 (5.91%)
Total	490,288	16,113 (3.29 %)	**1,714 (0.35%)**	10,629 (2.17%)	**117,801 (24.03%)**	243,698 (49.70%)	100,333 (20.46%)

Arch.	Total No. of Functions*	Control Flow Graph Recovery					
		F_s				F_m	S_a
		ANGR	GHIDRA	IDA	RADARE		
x86	121,108	10,622 (8.77%)	**75 (0.06%)**	375 (0.31%)	**26,209 (21.64%)**	48,146 (39.75%)	35,681 (29.46%)
x64	109,791	5,025 (4.58%)	**153 (0.14%)**	3,108 (2.83%)	**36,303 (33.07%)**	27,629 (25.17%)	37,573 (34.22%)
ARM	105,674	8,182 (7.74%)	91 (0.09%)	**79 (0.07%)**	**16,414 (15.53%)**	60,534 (57.28%)	20,374 (19.28%)
MIPS	126,179	**18,244 (14.46%)**	**53 (0.04%)**	64 (0.05%)	17,115 (13.56%)	71,988 (57.05%)	18,712 (14.83%)
Total	462,752	42,073 (9.09%)	**372 (0.08%)**	3,626 (0.78%)	**96,041 (20.75%)**	208,297 (45.01%)	112,340 (24.28%)

Table 3: Results of differential testing of various analysis. For function boundary detection and calling convention recovery, F_s and F_m indicate the number of binaries with single tool divergence (*i.e.,* only one tool produces a different result) and multi-tool divergence (*i.e.,* Multiple tools produce different results), respectively. S_a shows the number of times all the tools perfectly agreed with each other. For control flow graph recovery, F_s, F_m indicate the divergence for number of functions S_a indicates the number of times all tools agree on functions. (*) Functions in the randomly sampled binaries whose boundaries are correctly identified by all the tools.

Arch.	ANGR						GHIDRA						IDA						RADARE					
	N	A	S	E	P	T	N	A	S	E	P	T	N	A	S	E	P	T	N	A	S	E	P	T
x86	10,389	0	3	10	220	0	18	0	0	0	57	0	304	0	10	28	0	33	5,091	24	0	0	21,094	0
x64	3,863	0	12	16	1,134	0	94	0	7	0	52	0	2,844	4	87	173	0	0	6,246	84	0	0	29,973	0
ARM	7,678	0	0	0	233	271	6	0	0	0	85	0	23	0	11	23	22	0	8,920	30	45	0	7,419	0
MIPS	18,244	0	0	0	0	0	0	0	18	0	35	0	46	0	7	11	0	0	1,989	0	0	1	15,125	0
Total	40,174	0	15	26	1,587	271	118	0	25	0	229	0	3,217	4	115	235	22	33	22,246	138	45	1	73,611	0

Table 4: Detailed breakdown of the CFG results with single tool divergence. 'N' indicates a mismatch in the number of basic blocks. 'A' shows cases where the number of basic blocks match, but the starting addresses differ. 'S' indicates cases where the size(s) of the basic blocks do not match, 'E' indicates cases where the edges do not match, and 'P' indicates cases when the tools gave incomplete output. 'T' indicates cases for which the tool timed out.

to improve existing datasets, consequently helping in creating better models.

5 LIMITATIONS AND FUTURE WORK

Although, CORNUCOPIA is effective at generating a plethora of binaries. It has the following limitations.

Compiler bugs: We assume that the provided compiler preserves the semantics of the program in the generated binary. However, this may not be the case. The compiler may have bugs [61, 74] resulting in binaries that may not be semantically equivalent, especially those concerning undefined behavior.

Compiler frontend overhead: Although we mainly use the backend or code generation component of a compiler, in the general case, we unnecessarily run all components of the compiler, including its frontend. This adds a lot of overhead [41] as demonstrated by the relatively low yield by `gcc` (Section 4.3.2).

Arch		Prec		Rec		F1	
		R	O	R	O	R	O
x86	Name	62.6	**7.4**	62.5	**15.6**	62.5	**10.0**
	Type	63.7	**11.1**	63.7	**33.9**	63.7	**15.6**
	Overall	63.1	**9.3**	63.1	**24.0**	63.1	**12.9**
x64	Name	63.5	**3.2**	63.1	**5.2**	63.3	**3.9**
	Type	74.1	**24.7**	73.4	**47.8**	73.8	**31.9**
	Overall	68.8	**14.2**	68.3	**26.7**	68.6	**18.2**
ARM	Name	61.6	**7.0**	61.3	**12.5**	61.5	**8.7**
	Type	66.8	**14.6**	68.0	**24.8**	67.4	**17.9**
	Overall	64.2	**10.7**	64.7	**20.3**	64.5	**13.6**

Table 5: Figure showing Precision (Prec), Recall (Rec), and, F1 scores for Debin across 3 different architectures: The columns Reported (R) and Observed (O) show reported and observed scores on Cornucopia generated binaries.

Completeness of the Generated Dataset: Cornucopia uses existing programs to generate diverse binaries, and the completeness (*e.g.,* instructions covered in the underlying ISA) of the generated dataset depends on the features present in the corresponding programs. For instance, a program that does not use any floating point variables is unlikely to produce binaries with floating point instructions *e.g.,* `fcmovb`. This can be handled by using programs from diverse sources, such as Debian Repositories [25], GitHub, etc. We can also use systematic approaches such as Csmith [74] to generate C programs with the desired features and then use them in Cornucopia to generate a complete binary dataset.

Minimizing compiler crashes: Although, as discussed in Section 4.2.3, Cornucopia could find compiler crashes, it does not try to triage (*e.g.,* minimizing options and the target binary) them. We plan to integrate techniques like Delta Debugging [76] in our future work to minimize the set of crash-causing compiler flags.

6 RELATED WORK

Program obfuscation [47] is a well-known technique to change a program's structure without affecting the underlying functionality. One possible approach to the problem of this paper is to use various obfuscation techniques [37, 68] to generate semantically equivalent but structurally different binaries. Many initial techniques [17, 64] are aimed towards source or IR level obfuscation. tigress [18] is a source-to-source transformer that has various configurable transformations, such as control-flow flattening [39] and opaque-predicates [19]. Similarly, ob-llvm [33] enables applying a limited set of transformations at the LLVM IR level. Closure [43] uses stochastic optimization to select a sequence of transformations to produce the optimal obfuscation potency. Although these techniques are effective at modifying the program at IR or source code level, they have less impact on the generated binary [44].

A few binary-level techniques obfuscate control-flow using error handling semantics such as signals [51] and exception handling [71]. Other virtual machine–based techniques [24, 34] transform the given binary into a custom virtual machine. These binary-level techniques are known to introduce performance overhead [4]. The binary-level techniques are based on a fixed set of carefully designed patterns [47], which do not capture the entire range of behaviors of the underlying ISA. Finally, the primary goal of obfuscation techniques is to generate a hard-to-understand version of a given program [10]. In contrast, Cornucopia does not care about the understandability of the generated binary as long as it is different from all previously seen variations. The use of a compiler to generate binaries have been explored before, especially in the area of software diversity [30, 31, 38, 56]. These techniques only consider limited, non-performance-impacting transformations. Cornucopia has no such restrictions and explores all possible variations of the binary using compiler flags.

Although the effects of non-standard compiler optimizations on the generated binary have been explored before [12], the recent work BinTuner [53] is the most closely related to Cornucopia. However, as explained in Section 1, BinTuner requires considerable effort to use as it requires specifying conflicting compiler flags manually as first-order constraints. Cornucopia is completely automated and uses a feedback-guided approach to identify conflicting options automatically. Furthermore, as shown in Section 4.3, Cornucopia is more effective than BinTuner in efficiently generating diverse binaries. The use of fuzzing, especially AFL++, to generate a sequence of tokens has been explored before to fuzz interpreters [55]. Our approach allows the fuzzer to use its input generation ability fully, and enables Cornucopia to be easily configurable to use other fuzzers.

7 CONCLUSIONS

We present Cornucopia, an architecture, compiler agnostic and automated framework that generates a plethora of diverse binaries from program source code by using feedback-guided fuzzing. Our evaluation shows that Cornucopia is generally more effective at generating diverse binaries for a given program than BinTuner, a closely related work. It can be scaled on multiple threads for faster binary generation and better resource utilization. We showed that many binary analysis frameworks perform poorly on Cornucopia generated binaries opening up opportunities for more research in this area. We envision that Cornucopia becomes part of a binary analysis testing framework and helps in creating more robust analysis tools.

ACKNOWLEDGMENTS

We would like to thank all the anonymous reviewers for their valuable feedback and suggestions which helped in making this paper the best version of itself. This project was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract number N6600120C4031 and National Science Foundation (NSF) awards CCF-1908504 and CCF-1919197. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government, National Science Foundation or any agency thereof.

REFERENCES

[1] 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli

[2] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 70–83.

[3] Nadia Alshahwan, Earl T Barr, David Clark, George Danezis, and Héctor D Menéndez. 2020. Detecting malware with information complexity. *Entropy* 22, 5 (2020), 575.

[4] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. 2007. Program obfuscation: a quantitative approach. In *Proceedings of the 2007 ACM workshop on Quality of protection*. 15–20.

[5] Dennis Andriesse. 2018. *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly.* no starch press.

[6] Angr. 2021. Angr Issues. https://github.com/angr/angr/issues?q=is%3Aissue+error.

[7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

[8] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur, Kathryn E Gray, Prashanth Mundkur, Robert M Norton, Christopher Pulte, Alastair Reid, Peter Sewell, et al. 2018. Detailed models of instruction set architectures: From pseudocode to formal semantics. In *25th Automated Reasoning Workshop*. 23.

[9] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 845–860.

[10] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2001. On the (im) possibility of obfuscating programs. In *Annual international cryptology conference*. Springer, 1–18.

[11] Chandan Kumar Behera and D Lalitha Bhaskari. 2015. Different obfuscation techniques for code protection. *Procedia Computer Science* 70 (2015), 757–763.

[12] Craig Blackmore, Oliver Ray, and Kerstin Eder. 2017. Automatically tuning the gcc compiler to optimize the performance of applications running on embedded systems. *arXiv preprint arXiv:1703.08228* (2017).

[13] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. 2015. ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures. *ACM Transactions on Computer Systems (TOCS)* 33, 1 (2015), 1–34.

[14] Juan Caballero and Zhiqiang Lin. 2016. Type inference on executables. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–35.

[15] Pan Cake. 2017. Libre and Portable Reverse Engineering Framework. https://rada.re/n/.

[16] Chris Casinghino, JT Paasch, Cody Roux, John Altidor, Michael Dixon, and Dustin Jamner. 2019. Using binary analysis frameworks: The case for BAP and angr. In *NASA Formal Methods Symposium*. Springer, 123–129.

[17] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. 2009. The effectiveness of source code obfuscation: An experimental assessment. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 178–187.

[18] Christian Collberg. 2021. The tigress c obfuscator. https://tigress.wtf/.

[19] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 184–196.

[20] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S Adve, and Grigore Roşu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1133–1148.

[21] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489.

[22] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of executable objects (english version). *Sstic* 5, 1 (2005), 3.

[23] Michael Eager. 2012. The DWARF Debugging Standard. https://dwarfstd.org/.

[24] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. 2011. Multi-stage binary code obfuscation using improved virtual machine. In *International Conference on Information Security*. Springer, 168–181.

[25] José Angel Galindo, David Benavides, and Sergio Segura. 2010. Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis.. In *ACoTA*. Citeseer, 29–34.

[26] Frederic Grelot, Sebastien Larinier, and Marie Salmon. 2021. Automation of Binary Analysis: From Open Source Collection to Threat Intelligence. *Proceedings of the 28th C&ESAR* (2021), 41.

[27] Aric Hagberg and Drew Conway. 2020. NetworkX: Network Analysis with Python.

[28] Nadav Har'El. 2017. x86 Floating point exceptions - incorrect support? https://bugs.launchpad.net/qemu/+bug/1668041.

[29] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1667–1680.

[30] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. 2018. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology* 104 (2018), 72–93. https://doi.org/10.1016/j.infsof.2018.07.007

[31] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. 2011. Compiler-generated software diversity. In *Moving Target Defense*. Springer, 77–98.

[32] Haegeon Jeong, Jeanseong Baik, and Kyungtae Kang. 2017. Functional level hot-patching platform for executable and linkable format binaries. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 489–494.

[33] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM–software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 3–9.

[34] Patrick Kochberger, Sebastian Schrittwieser, Stefan Schweighofer, Peter Kieseberg, and Edgar Weippl. 2021. SoK: Automatic Deobfuscation of Virtualization-protected Applications. In *The 16th International Conference on Availability, Reliability and Security*. 1–15.

[35] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*. IEEE, 533–537.

[36] Jesse Kornblum. 2006. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation* 3 (2006), 91–97.

[37] Pengwei Lan, Pei Wang, Shuai Wang, and Dinghao Wu. 2017. Lambda obfuscation. In *International Conference on Security and Privacy in Communication Systems*. Springer, 206–224.

[38] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 276–291.

[39] Tımea László and Ákos Kiss. 2009. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30, 1 (2009), 3–19.

[40] lcamtuf. 2014. Binary fuzzing strategies: what works, what doesn't. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html.

[41] David Xinliang Li, Raksit Ashok, and Robert Hundt. 2010. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 53–61.

[42] Yan Lin and Debin Gao. 2021. When Function Signature Recovery Meets Compiler Optimization. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–52.

[43] Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. 2017. Stochastic optimization of program obfuscation. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 221–231.

[44] Matias Madou, Bertrand Anckaert, Bruno De Bus, Koen De Bosschere, Jan Cappaert, and Bart Preneel. 2006. On the effectiveness of source code transformations for binary obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP06)*. CSREA Press, 527–533.

[45] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 309–329.

[46] Dongyu Meng, Michele Guerriero, Aravind Machiry, Hojjat Aghakhani, Priyanka Bose, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2021. Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 731–743.

[47] Jasvir Nagra and Christian Collberg. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Pearson Education.

[48] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 180–190.

[49] 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. *2021 IEEE Symposium on Security and Privacy (SP)* (2021), 833–851.

[50] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 690–702.

[51] Igor V Popov, Saumya K Debray, and Gregory R Andrews. 2007. Binary Obfuscation Using Signals.. In *USENIX Security Symposium*. 275–290.

[52] randare. 2021. Radare2 Issues. https://github.com/radareorg/radare2/issues?q=is%3Aissue+error.

[53] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: an empirical

study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 142–157.

[54] Rolf Rolles. 2014. COMPILER OPTIMIZATIONS FOR REVERSE ENGINEERS. https://www.msreverseengineering.com/blog/2014/6/23/compiler-optimizations-for-reverse-engineers.

[55] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*. 2795–2809.

[56] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software diversity: state of the art and perspectives.

[57] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[58] Jagsir Singh and Jaswinder Singh. 2021. A survey on machine learning-based malware detection in executable files. *Journal of Systems Architecture* 112 (2021), 101861.

[59] Vidush Singhal, Akul Abhilash Pillai, Charitha Saumya, Milind Kulkarni, and Aravind Machiry. 2022. Cornucopia: A Framework for Feedback Guided Generation of Binaries. https://doi.org/10.48550/ARXIV.2209.06694

[60] YN Srikant and Priti Shankar. 2018. *The compiler design handbook: optimizations and machine code generation*. CRC Press.

[61] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 294–305.

[62] Peter Szor and Peter Ferrie. 2001. Hunting for metamorphic. In *Virus bulletin conference*. Prague.

[63] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. 2020. Sound C Code Decompilation for a subset of x86-64 Binaries. In *International Conference on Software Engineering and Formal Methods*. Springer, 247–264.

[64] Alessio Viticchié, Leonardo Regano, Marco Torchiano, Cataldo Basile, Mariano Ceccato, Paolo Tonella, and Roberto Tiella. 2016. Assessment of source code obfuscation techniques. In *2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM)*. IEEE, 11–20.

[65] Chang Wang, Zhaolong Zhang, Xiaoqi Jia, and Donghai Tian. 2018. Binary Obfuscation Based Reassemble. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*. 153–160. https://doi.org/10.1109/MALWARE.2018.8659363

[66] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. 2015. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference*. 331–340.

[67] 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*.

[68] Yan Wang, Shuai Wang, Pei Wang, and Dinghao Wu. 2017. Turing obfuscation. In *International Conference on Security and Privacy in Communication Systems*. Springer, 225–244.

[69] 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.

[70] Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. 2010. Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the 17th ACM conference on Computer and communications security*. 536–546.

[71] Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. 2010. Mimimorphism: A New Approach to Binary Code Obfuscation. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. Association for Computing Machinery, New York, NY, USA, 536–546. https://doi.org/10.1145/1866307.1866368

[72] Liang Xu, Fangqi Sun, and Zhendong Su. 2009. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep* (2009).

[73] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. 2019. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access* 7 (2019), 65889–65912.

[74] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[75] Eric Youngdale. 1995. Kernel korner: The ELF object file format: Introduction. *Linux Journal* 1995, 12es (1995), 7–es.

[76] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[77] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 813–832.

[78] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–31.