FUZZING DEEPER LOGIC WITH IMPEDING FUNCTION TRANSFORMATION

by

Rowan Hart

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

Department of Computer Science

West Lafayette, Indiana

December 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Antonio Bianchi, Chair

School of Computer Science

Dr. Berkay Celik

School of Computer Science

Dr. Dave (Jing) Tian

School of Computer Science

Approved by:

Dr. Kihong Park

Dedicated to Kerry Brock, who brought me old printers for Christmas and set me on the path towards science, and to Evelyn Doster, who made this possible from beginning to end.

ACKNOWLEDGMENTS

Xingman Chen, Alex Lin, and Connor McMillin conceptualized much of what would become this work, and Michael Tompkins took equal part in completing it. Thank you also to Antonio Bianchi and Fish Wang for their advice, guidance, and expertise.

Special thanks to Bader AlBassam, who introduced me to computer security and CTF, and has lent a hand every step of the way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Fuzzing, a technique for negative testing of programs using randomly mutated or generated input data, is responsible for the discovery of thousands of bugs in software from web browsers to video players. Advances in fuzzing focus on various methods for enhancing the number of bugs found and reducing the time spent to find them by applying various static, dynamic, and symbolic binary analysis techniques. As a stochastic process, fuzzing is an inherently inefficient method for discovering bugs residing in *deep logic* of programs due to the compounding complexity of preconditions as paths in programs grow in length. We propose a novel system to overcome this limitation by abstracting away path-constraining preconditions from a statement level to a function level by identifying impeding functions, functions that inhibit control flow from proceeding. REFACE is an end-to-end system for enhancing the capabilities of an existing fuzzer by generating *variant binaries* that present an easier-to-fuzz interface and expands an ongoing fuzzing campaign with minimal offline overhead. REFACE operates entirely on binary programs, requiring no source code or symbols to run, and is fuzzer-agnostic. This enhancement represents a step forward in a new direction toward abstraction of code that has historically presented a significant barrier to fuzzing and aims to make incremental progress by way of several ancillary dataflow analysis techniques with potential wide applicability. We attain a significant improvement in speed of obtaining maximum coverage, re-discover one known bug, and discover one possible new bug in a binary program during evaluation against an un-modified state-of-the-art fuzzer with no augmentation.

1. INTRODUCTION

1.1 Prior Work & History

Fuzzing, static and dynamic analysis, and symbolic execution fields have deep prior work to draw on and learn from when considering approaches for enhanced bug finding. Our approach draws on all of the following sub-fields of binary security research, therefore it is important to understand the strengths, weaknesses, and direction of each. At their core, every method has the goal of finding defects in software by improving in some aspect of analysis, but there are highly varied methods for approaching the problem.

1.1.1 Static Analysis

The concept of "static analysis" is broad, but can generally be defined as any software analysis that does not execute the code being analyzed. Common modern static analyses are Control Flow Graph (CFG) construction, variable recovery, and decompilation. The first decompiler implementing these techniques was written by Housel [1] at Purdue university, a very rough prototype leveraging an Intermediate Representation (IR), CFG construction and analysis, as well as dead code elimination and optimization. Mention of static analysis by name in literature first occurs circa 1975 with a focus on data flow of programs [2] for both software reliability [3] and program security [4]. The bulk of this work was focused on FORTRAN [5] programs. Simultaneously, Cousot's field-defining paper which originated the term "Abstract Interpretation" [6] was released. This paper defined several critical constructs to program analysis, notably the use of a lattice-based approach for analyzing program flows. Their assertion that program context is a complete lattice is foundational for modern compiler techniques implemented in frameworks such as LLVM [7] as well as program and binary analysis tools such as MAYHEM [8] and BAP [9]. Abstract intepretation formed a basis for future static analysis techniques as a generalization of several methods for analyzing programs, including global data flow analysis, type verification, type discovery, symbolic evaluation, and verification of program correctness.

Historical Methods

These foundational static analysis techniques focused primarily on source code. From 1970-90, this was primarily FORTRAN and PASCAL code. By the mid 1990s, static analysis of binary programs had become a prominent topic. In particular, Cifuentes [10] pioneered data and code flow based decompilation approaches in 1995, unifying several static analysis techniques into a useful C decompilation tool `dcc`. Cifuentes also pioneered several other areas. Slicing leverages data and control dependency analysis to create a minimal set of binary instructions, or *slice* of a program either terminating or beginning at a particular code location. Soon after her work on decompilers, Cifuentes published a technique [11] for performing slicing *intra-procedurally*, or within one function. Landi in 1992 [12] expanded the area of *alias* analysis, proving that *context-sensitive* static analysis is an undecidable problem and at best NP complete under many simplifying assumptions. As Landi puts it, "this is an extremely negative result", one that the last two decades of static analysis research have focused on. Despite provable properties about the processes that make up static analysis of programs, heuristics are able to solve many problems in a vast majority of cases in static analysis.

In the late 1990s thorough approximately 2005, there was significant effort in heuristic and case-specific methodologies for analyzing binary programs. The two main frameworks for analysis, Abstract Interpretation and Dataflow Analysis had been solidified enough to become generally useful, so additional effort was made to solve less broad, more practical problems. CFG construction was greatly expanded by Theiling [13] with the introduction of bottom-up and iterative algorithms for construction and enhancement of graphs without relying on symbols or debug information. One motivation for this is the accelerated spread of malware [14] at this time, the analysis of which necessitated advancements in decompilation, debugging, and static analysis. Compounding this need is the lack of availability of dynamic techniques for malware analysis prior to the wide introduction of Virtual Machines (VMs) [15]. Cifuentes et al. [16] implemented a generalized technique for recovering *jump tables* in binaries using slicing and constant propagation. Further enhancements to slicing by Kiss et

al. [17] allowed size reductions of binaries under analysis, speeding up further manual static analysis and enabling slicing *inter*-procedurally, building on the earlier work of Cifuentes. Harris et al. [18] focused on enhancing the ability of static analysis tools to handle *stripped* binaries where symbols and debug information have been completely removed. This period marked growth in specific areas, which would eventually be merged to create new and more capable tools for researchers and industry.

Malware-motivated Advancement

In 2007, Hex-Rays released the decompiler module for IDA Pro [19]. This decompiler was the first example of a very widely adopted, production quality decompilation engine and incorporated many of the recent enhancements to the static analysis process. Guilfanov wrote "it heavily uses the data-flow analysis methods to analyze the program" and envisioned a future where it "is capable of answering questions about the variable ranges, code and data coverage...check if some invariants hold at the given program locations...report about not only trivial buffer overflows but also other logic flaws". Guilfanov would prove prophetic: the next decade gave rise to rapid innovation in binary static analysis, in particular practical applications of existing methods to new and continuing problems: malware and ever expanding technology and internet sectors. Kruegel et al. [20] introduced several techniques for improving analysis of stripped binaries. The correlation between papers focusing on malware analysis and focus on stripped or obfuscated binaries also lends itself to the eventual goal of many modern binary analysis frameworks: to make less assumptions about the target binary. Less assumptions about the binary means a higher requirement to correctly and safely analyze it, but allows the techniques used to be more generally applicable. This time period also saw greatly increased interest in use of static analysis for vulnerability detection. Static Application Security Testing (SAST) tools existed for quite some time, but were highly focused on auditing source code, not binary applications. Cova et al. [21] presented a groundbreaking work that leveraged static and symbolic techniques to find vulnerabilities in binaries in 2006. We focus here on the static aspects of this approach, and discuss the

symbolic aspects in Subsection 1.1.2. Cova et. al use primarily linear sweeps to discover code in binary blobs in their work. Call and jump resolution, including jump table resolution resembling the algorithm given by Cifuentes, and constant propagation also put forth by Cifuentes are used to construct a highly accurate CFG. Cova et al. go a step further to refine the CFG by detecting loops and recursive functions. Finally, they use the binary's structures to detect library function use. The core of their approach to finding vulnerabilities centers on using symbolic execution to add *program context* to the static view of the binary given by the CFG. This work was one of the first to both merge many analysis techniques into a single tool, and one of the first to attempt analysis for vulnerability discovery across many vulnerability categories.

Several other works such as BitBlaze [22] and Jakstab [23] attempted a similar task, each with various implementation differences and goals. For example, BitBlaze implemented a new *weakest-precondition* analysis that allows for correctness proofs about program behavior to be tested statically. BitBlaze focused primarily on correctness assertions about behavior than on direct bug-finding, but reached the same result: incorrect behavior is indeed a bug. Jakstab adopted a less academic approach and instead aggressively optimized lifted IR code to produce the smallest possible CFG to analyze. CodeSurfer/x86 [24] debuted two highly impactful static analysis techniques. The first, *value-set analysis* (VSA), first put forth by Balakrishnan et al. efficiently tracks "an over-approximation of the set of values that each data object can hold at each program point" allowing fast and accurate data-flow analysis. More importantly, this analysis is context-sensitive. Codesurfer/x86 also utilized the idea of affine relationships to determine additional information about the value-sets of registers at code points. Finally, CodeSurfer/x86 was one of the first projects along with the work by Cova et al. to implement indirect jump and call resolution; a key step in inter-procedural analysis. Without indirect call resolution, the target of any call to a location residing in a register or memory value cannot be determined, and propagation of information through a call stack or via context sensitive path operations is much more difficult or less accurate.

Compiler-based and Satisfiability Theory

The Cyber Grand Challenge [25] (CGC) and the new concept of the "Cyber Reasoning System" (CRS) gave rise to many tools that leveraged and improved existing static analysis techniques as well as facilitated the creation of entirely new techniques. The winner of CGC, Mayhem [8] , was based on the BAP [9] tool, due to Brumley et al. BAP implements a much less recent technology, syntax-directed analysis. This analysis was discounted in the 80s due to the difficulty of representing all effects of an instruction accurately. BAP solved this problem by representing all side effects explicitly. It provides most of the previously mentioned static analysis tools such as CFGs and VSA. Like its predecessor BitBlaze, BAP places great emphasis on the *weakest precondition* analysis problem, but extends prior functionality via Satisfiability Modulo Theory (SMT) solvers. These solvers allow greater flexibility of formal verification of code functionality. BAP also implements SSA forms in its Intermediate Language (IL), a technique in which values are assigned to only once. This was not a novel idea, SSA form and algorithms for value numbering are due to Cytron [26] and most recently Braun [27]. BAP implements algorithms for the latter including an algorithm for ϕ-node reduction using Strongly Connected Components (SCCs) atop the CFG. This method of value numbering is highly advanced and efficient, but relies on solid underpinnings of CFG consistency and completeness. The viability of this approach demonstrates the leaps and bounds of static analysis evolution to this point.

Static analysis research has slowed as of 2022. Binary application security research has pivoted in directions that may in fact be static but do not fit the prior trends. Use of AI for static analysis by developing embeddings such as code2vec [28] and creating feature selection models [29] bears little resemblance to current formal-methods based approaches, yet it presents an exciting direction to expand capabilities through innovation in other areas. Other research seeks to refine existing technologies, such as the BinCAT [30] project to leverage older Abstract Interpretation techniques to implement bit-level precision for value analysis. Blending of binary analysis technologies began to accelerate with CGC-adjacent projects. For example, modern analyses such as Veritesting [31] leverage static analysis dur-

ing dynamic symbolic dynamic analysis to enhance efficiency and accuracy. Static analysis still forms the core of binary analysis, and will for the foreseeable future.

1.1.2 Symbolic Execution

Symbolic execution is a sub-category of abstract interpretation, and is a set of methods for determining information about possible executions of a program. Symbolic execution was first proposed in the 1970s by King [32] [33] for the purpose of verifying the correctness of programs. Essentially, the technique translates the sub-expressions of each instruction executed in a program into a symbolic expression, the result of which may take on many possible values. Likewise, control flow transfers that depend on these expressions is evaluated such that all possible paths may be analyzed. For example, consider the small C program in Listing 1.

```
1  int main() {
2      int32_t val;
3      read(0, &val, sizeof(val));
4
5      if (val < 0) {
6          printf("Too small!\n");
7          exit(1);
8      }
9
10     if (val > 10000) {
11         printf("Too big!\n");
12         exit(1);
13     } else if (val > 1000) {
14         printf("Victory!\n");
15         exit(0);
16     } else {
17         printf("Too small!\n");
18         exit(1);
19     }
20 }
```

Listing 1: Example of a simple program for symbolic execution

The variable `val` declared on line 2 under symbolic execution would be represented as a symbolic variable of size 4 bytes, which we will call `V(val)`. The value is set via a `read` call, which takes user input. In symbolic execution, user input need not be provided, and the value may be left as a purely symbolic value. This means `V(val)` may take on any value that can be represented in its 4 byte size. At line 5, `val` is compared against 0, and a branch is taken depending on that comparison. Herein lies the crux of symbolic execution: instead of assigning a nonnegative or negative value to `val`, the state of the execution will be duplicated. In one state, the branch is taken, and thus `V(val)` must have a negative value. This is represented by applying a *constraint* that $V(val) < 0$ to the "split state" which allows the state to embody restrictions on the data it contains while not assigning any specific value to the symbolic variable `V(val)`. In the second state, the branch is not taken, and a different constraint $V(val) \geq 0$ is applied. This process continues forward from the program start point, and after execution of this example four states will exist with various constraints and "deadend" locations:

1. $V(val) < 0$, deadends at line 7.
2. $V(val) \geq 0 \wedge V(val) > 10000$, deadends at line 12.
3. $V(val) \geq 0 \wedge V(val) \leq 10000 \wedge V(val) > 1000$, deadends at line 15.
4. $V(val) \geq 0 \wedge V(val) \leq 10000 \wedge V(val) \leq 1000$, deadends at line 18.

Human-directed systems

Various techniques exist to process and simplify constraints, notably the use of *Satisfiability Modulo Theory* (SMT) solvers such as Z3 [34], but early executors relied on more simple approaches. In this contrived example, the constraints $V(val) \leq 10000 \wedge V(val) \leq 1000$ could be simplified to simply $V(val) \leq 1000$. Using their symbolic execution engine EFFIGY, King [35] points out that symbolic execution can be viewed as a tree of possibilities beginning from a program's entrypoint. However, EFFIGY was very simple, required human interaction, and lacked a complex memory model. Despite these limitations, it paved the way for future work such as SELECT [36]. SELECT used a similar model of interactive

debugging of LISP programs to achieve formal verification of small parts of symbolically executed programs. It allows a user to specify "assertions" in the program that can then be verified symbolically. For example, in the above example in Listing 1, an assertion that execution will always terminate with a call to `exit` could be verified. From a security perspective, assertions about insecure conditions such as buffer overflows, out of bounds reads or writes, or other exploitation primitives are most interesting. Unlike EFFIGY, SELECT implements a feature that most modern security-oriented symbolic execution systems inherit: input data generation. By solving the constraints placed on input data, SELECT is able to return an input that causes program execution to proceed down a particular path, passing specific checks and taking specific branches. Notably, SELECT is only able to solve for inputs that can be expressed as simple systems of linear equations, and cites "technical limitations in present-day theorem-proving techniques" as a major obstacle to effective use of symbolic execution. SELECT introduced several other key ideas that would inform symbolic execution research in the future. First, recognition of the state explosion problem. State explosion occurs when repeated branching causes an exponential increase in the number of active program states. A common pattern leading to state explosion presented by the SELECT authors is a loop that executes a symbolic number of iterations. SELECT handles this by setting a maximum number of iterations, but the authors recognize this as an imperfect solution. Next, they present an idea of what is now commonly referred to as "symbolic procedures" where a known function can be skipped during symbolic execution. They suggest an "attempt to characterize each subroutine invoked...whenever that module is invoked it can be replaced...by those input/output specifications.". DISSECT, due to Howden [3] is stated by the author to be similar to SELECT, except that it presents a programming API for writing "simple analysis procedures" over the programs it is executing.

Abstract methods

Cousot [6] et al. mention in their field-defining work on Abstract Interpretation is a superset of symbolic evaluation, among other symbolic techniques. Abstract Interpretation

is a process by which a program's real representation is transformed in a (usually simplified) representation where properties may be computed efficiently and information about the real program can be obtained. Fundamentally, the technique seeks to represent a semantic property of a program via set relations from a concrete set to an abstract one. In a later simplified presentation by Cousot [37], the idea is presented more succinctly as representing program executions as "trajectories" then utilizing Abstract Interpretation to find an abstraction of the program semantics that covers all "trajectories". Finally, the abstraction is checked against various error conditions such as NULL pointer dereferences in a process known as "bounded model-checking". The example provided in the original paper describes a useful program analysis technique. Given a program P, consisting of instructions $I_{1...n}$, transform each aritmetic operation on a number into a simpler arithmetic operation on the "sign" of the number. In Cousot's example, $-1515 \cdot 17 \implies -(+) \cdot (+) = (-)$, i.e. the result of this computation *must* be negative. In a program where a negative result could create an error condition, for example a negative array index, this Abstract Interpretation could allow for bug checking in a significantly more efficient manner. A key property of Abstract Interpretation is the "fundamentally incomplete' results...allows the programmer or the compiler to answer questions which do not need full knowledge of program executions". In nearly every paper concerning symbolic execution after Cousot's work, Abstract Interpretation is mentioned as a superset of symbolic execution.

A follow-up to their earlier paper on EFFIGY, King et al. [38] expands on several previously glossed-over concepts. They reiterate that generally, symbolic execution performs symbolic operations during evaluation of symbolic expressions and during conditional branching. They also note that symbolic expressions are *created* during any input to the program, and expand on earlier definitions of input to include uninitialized values, input from file user input or networks, randomness, and more. The authors identify the idea of *exhaustive* symbolic execution, where a program is completely explored and every possible state in the program is identified. They note that this is an extremely intensive process over all but the simplest programs, as it requires not only exploring all possible control flow paths in the program, but every control flow point with *each possible preceeding path* as

well. Symbolic execution experienced relatively few advancements after this until the early 2000s, seemingly primarily due to the lack of advancements in efficient theorem proving that made the relatively ineffective linear systems solvers leveraged by SELECT and EFFIGY less effective than necessary to tackle "realistic" programs.

SMT and Boolean Satisfiability solvers

Microsoft Research (MSR) led the advancement of symbolic execution advancement starting with Ball [39]'s 2003 paper. Ball created several key advancements in symbolic execution that are still considered state of the art. They analyze C programs, as opposed to the earlier systems discussed which analyze ALGOL, FORTRAN, and other similar languages. To understand the significance of this choice, we must first understand the aliasing problem. *Aliasing* occurs in languages with *pointers*, where a value refers to a location in memory, when multiple *pointers* refer to the same location in memory. This presents problems when, for example, one pointer is used to write to the referenced memory. The other pointer has no "knowledge" that this has occurred, and in an analysis setting it is difficult to determine as well. C also allows pointers to code, as well as multiple layers of indirection, which both compound the aliasing issue. Finally, C is the *lingua franca* of modern systems, and is often used as an additional Application Binary Interface (ABI) or as a wrapper to external ABIs. Ball also introduced the use of boolean satisfiability as an abstraction for systems, a method reminiscent of Cousot's Abstract Interpretation, instead of using simple linear equations. To facilitate this, Ball employs Satisfiability (SAT) solvers, a solution now employed in every usable symbolic execution engine. Ball's system uses three steps to accomplish another critical goal of symbolic execution: full automation. Whereas early systems like SELECT required, according to the author, frequent intervention by a human analyst, Ball's system is able to accomplish the following in an automated fashion.

First, given a C program P, is transformed into a *boolean program* BP in which "every feasible execution path of P is a feasible execution path of BP". We examine an example buggy program P and its BP equivalent in Listing 2 from Ball's paper.

```c
// P
void partition(int *a, size_t n) {
    int pivot = *a;
    int tmp = -1;
    size_t lo = 1;
    size_t hi = n -1;

    while (lo <= hi) {
        // missing check:
        // lo <= hi
        while (*(a + lo) <= pivot) {
            lo++;
        }

        // missing check:
        // lo <= hi
        while (*(a + hi) > pivot) {
            hi--;
        }

        if (lo < hi) {
            tmp = *(a + lo);
            *(a + lo) = *(a + hi);
            *(a + hi) = tmp;
        }
    }
}
```

```c
bool check(bool a, bool b) {
    if (a) {
        return true;
    } else if (!a && b) {
        return false;
    } else {
        return undetermined;
    }
}

// BP
void partition(int *a, size_t n) {
    bool lt = true;
    bool le = true;
    bool al = undetermined;
    bool ah = undetermined;

    while (le) {
        while (al) {
            lt = check(false, !lt);
            le = check(lt, !lt || !le);
            al = undetermined;
        }

        while (ah) {
            lt = check(false, !lt);
            le = check(lt, !lt || !le);
            ah = undetermined;
        }

        if (lt) {
            al = !ah;
            ah = !al;
        }
    }
}
```

Listing 2: Example of a program P and its boolean program equivalent BP

This methodology works by translating the boolean relationships (such as `lo <= hi`) into boolean variables, which can then be solved for by a boolean SAT solver to determine the set of reachable states and the values of each boolean variable that allows those states to be reached. They use reachability to narrow the amount of actual symbolic execution that must be performed, as the constraint solving during symbolic execution is the most expensive part of the execution process. Finally, they perform symbolic execution to explore the tree of possible states, similar to the idea put forth by King. From this process, they obtain both a *feasibility* result, which describes the full set of paths that can actually execute, and a set of inputs that cause each feasible path to execute. They use these feasible paths to identify that due to missing bounds checks, there are paths that result in an error state (which in this case results in an array that is not partitioned correctly). The input generation enables real execution testing of this error condition to more easily fix the problem. In addition, the transformation to boolean variables allows testing of this program, which has nested loops and comparisons, to avoid state explosion by computing relationships as invariant at each program point.

Ball's work was a huge step forward for symbolic execution methods, but had a large caveat of requiring significant high level program information and formal specification of error conditions of the code under test. Balakrishnan and Reps [40] cite the "growing need to tools that analyze executables" as a primary motivation in their work on *value-set* analysis, a method described as static that nonetheless contributed many ideas to symbolic techniques covered further on. As mentioned previously, the largest problem for symbolic execution is state explosion, Value-set analysis creates an abstraction of a program's data into value-sets, which describe the data more efficiently, although it provides an over-approximation. It also introduced the use of widening and narrowing operations on memory accesses to abstract memory locations as well as registers, a technique used by more modern analysis engines.

Scaling symbolic execution

CUTE [41] was one of the first *concolic* execution engines, a strategy that will be discussed later in Subsection 1.1.4. It used a novel approach to symbolic execution that was tailored to find bugs as opposed to earlier systems that were primarily concerned with correctness checking. Similarly to Ball's work, CUTE uses symbolic analysis of constraints on reachable paths in the program under test. It has two key differences, however. First, CUTE only observes constraints along a specific path triggered by an existing input, rather than explore the program's state space exhaustively. Second, instead of discovering new paths forward via exploration, it only observes the path that was already executed. Once the constraints are observed, CUTE first chooses a path constraint and inverts it. This creates a new path that will branch at a particular point in the program's execution. It then uses this new set of path constraints – including the just-inverted constraint – and solves it to identify an input that will trigger this new path. This method is known as *Dynamic Symbolic Execution* (DSE) or *dynamic symbolic tracing*, as it uses the symbolic components of "pure" symbolic execution to observe a dynamic execution of a program. Interestingly, CUTE does not use boolean satisfiability as an abstract representation of the program. Instead, it uses a linear constraint solver and falls back to using concrete values from DSE instead of all symbolic variables when the constraint solver is unable to solve or simplify a variable. The CUTE authors present several additional optimizations to linear solving. It uses heuristics to check whether it needs to run the solver at all, which they claim reduces solves by 60-95% depending on the program. They also eliminate common sub-expressions of constraints and uses an "incremental" approach to solving. All of these optimizations are present in modern well-supported solvers, but were novel ideas for their time. It is important to remember that constraint solvers have essentially been used orthogonal to their intended purpose when applied to program analysis. Finally, CUTE applies pointer analysis techniques in an attempt to mitigate the aliasing problem, although it should be noted that their approach requires source code.

Shortly after CUTE, Xie et al. presented Saturn [42], a system that uses prior ideas from other symbolic execution systems at larger scales and on realistic programs. Saturn makes several notable contributions. First, it applies the use of SAT solvers to analyzing locking errors, a notoriously difficult class of bug to analyze. As the authors point out, "locking is always a flow sensitive and sometimes a path sensitive property, programmers store locks in data structures, pass locks as arguments, use complex tests to decide when and where to acquire and release locks, and so on". Saturn performs symbolic execution in a very similar way to Ball, but augments the translated code with control flow graph information. This allows further constraint of possible control flow transfer locations following a just-executed block of code. The system primarily aims to reduce the number of symbolic queries to the SAT solver, and notes as prior research does that this is the most expensive component of the symbolic analysis process. Also unlike the work due to Ball, Saturn is able to perform analysis interprocedurally. For many functions, it uses a function summary to model the behavior of a function without performing actual expensive symbolic execution of the function. This greatly improves the speed of symbolic execution by reducing the amount of actual symbolic execution that needs to take place. In addition, the analysis is made path sensitive by incorporating the state history. The authors note that, for example, a `try_lock` operation is allowed to fail, and whether it does depends on the current state of the lock. Following a failure to lock in a `try_lock` operation, an error condition may or may not be encountered, and prior state is required to determine whether in fact an error condition is present. Saturn claims that it was "the most successful bug detection tool for Linux locking errors" at the time of publication.

Engler et al. introduced *under-constrained* symbolic execution [43] to tackle scalability problems with symbolic execution. *Under-constrained* symbolic execution makes one small but impactful change to typical symbolic execution such as that done by CUTE. A symbolic variable is either "normal" or "under-constrained" and errors involving "under-constrained" variables are only reported if the solver is able to show that the error occurs for all values of that variable. Otherwise, a new constraint that the variable does not take on an error-case value is added to the solver and execution continues. This coerces symbolic execution from

what is known as a *may* analysis, where any state that can possibly violate some assertion is in error, to a *must* analysis, where only states that are guaranteed to violate the assertion are in error. The authors implemented a tool called EXE that performs this *under-constrained* execution and used it to analyze Linux kernel drivers *in isolation* – a very interesting application of symbolic execution to an existing domain, but in a novel way.

Modern symbolic systems

One of the most famous symbolic execution engines is KLEE [44] due to its effective, fully automated approach to symbolic execution and bug finding. KLEE was presented by the authors of EXE, and expands on previous work in several ways. First, the authors use a "compact state representation" to mitigate the problem of state explosion not by reducing the number of states, as many other works attempt to do, but by reducing the size of each state to allow more states to fit in memory. This is an important contribution, and represents a change in thinking with respect to state explosion where it seems to become clear that the issue cannot be effectively mitigated by path pruning alone. This state representation uses copy-on-write memory for heap objects, not simply memory pages, to decrease the amount of memory that must be copied when states split. Like previous works, the KLEE authors place significant emphasis on the cost of constraint solving. KLEE uses the STP [45] solver with several optimizations based on compiler theory as well as caching. *Expression rewriting* for common arithmetic operations such as converting power of 2 to left shift, constant folding, and more. In addition, KLEE utilizes optimization of constraints to eliminate redundant constraints on paths. In the authors' example, if a variable has a constraint such as `j < 10`, and subsequently has another constraint such as `j == 5`, the first constraint can be eliminated. The authors claim a massive 95% reduction in the number of queries to the constraint solver, which reduces the execution time commensurately. Finally, KLEE presents a number of non-theoretical improvements to the engineering implementation of constraint solving systems. In particular, the authors pay attention to the simulated execution environment including system calls, the filesystem, and other external interactions.

KLEE allows mixed concrete and symbolic data in files, which permits more complex system configurations than other methods that either only use concrete data and perform DSE, or that use only symbolic data and experience a greater slowdown due to path explosion. The largest drawback of KLEE's implementation is the reliance on LLVM bytecode, which in most non-trivial cases means that KLEE only works when source code is available.

SAGE [46], presented near the same time, is also a whitebox vulnerability finding system that uses symbolic execution alongside fuzzing, which will be discussed in Subsection 1.1.4. Unlike KLEE, SAGE does not rely on information from source code to work, and instead uses program tracing to generate path constraints. It is unspecified what tool SAGE uses to perform tracing, but the authors note that the real difficulty in symbolic tracing arises from the complexity of the underlying machine instructions. This is particularly problematic on x86 platforms, and SAGE solves it by simply attempting to handle every case of the x86 instruction set exhaustively. SAGE also implements path constraint optimizations very similar to those of KLEE. S2E [47] was another project with similar goals that achieved great success by utilizing LLVM, like KLEE, and QEMU [48] to apply *dynamic binary translation* to scale symbolic execution. While the primary contribution of S2E was engineering effort, they made a significant contribution in recognizing that symbolic execution need not always be active, hence *selective* symbolic execution. By selecting when to enable and disable symbolic analysis, the expensive process can be restricted to a small area, allowing the rest of the system under test to execute efficiently. While KLEE and SAGE pioneered the application of symbolic execution to truly realistic binary programs, MAYHEM [8] brought the use of symbolic execution to the mainstream. MAYHEM incorporates many advances. First, it utilizes Microsoft Research's Z3 as a symbolic engine [34], a decision that is now taken for granted as Z3 incorporates a huge number of heuristics for very efficient expression simplification, especially over *bitvectors*, the symbolic data type typically used to model bits and bytes of memory. Next, MAYHEM uses a large number of heuristics for path selection. This is not necessarily a novel idea, but it reflects the reality that no algorithm will be able to efficiently choose the best path in all cases. Instead, MAYHEM attempts to implement the intuition that a human reverse engineer might use to analyze a binary to find likely bug loca-

tions. Next, the authors implement a unique memory model that makes all writes concrete, but allows reads to be partially or fully symbolic. They also implement the aforementioned *value-set analysis* in MAYHEM as a "preprocessing step" to utilizing Z3 to solve constraints to determine memory locations, an especially expensive task. Finally, MAYHEM attempts to determine whether expressions may be pointers to other data, and infers whether those pointers may be corruptible. This is the final extremely unique feature of MAYHEM: after determining that a bug may exist, MAYHEM attempts to synthesize a full exploit for the program to prove its vulnerability.

This "full-chain" approach to using symbolic execution to discover bugs also appears in two papers [49] [50] by the *angr* authors. The authors collect in essence, all previous work in symbolic execution into a single extensible library that can be used to provide nearly any of the previously discussed symbolic execution tactics. Of particular note is the careful attention to indirect jump resolution, the multi-modal memory model, and the reasonably complete system state capability for representing files and other external inputs. *angr* implements DSE, Veritesting, under-constrained DSE, fuzzing with symbolic execution, static analysis of control and dataflow, symbolic program states and environment simulation, decompilation, forward and backward symbolic execution and slicing, integration with various levels of full system and userspace emulation, taint analysis, and more. Implementations of more cutting-edge approaches are typically added to *angr* by the authors after their introduction, so it represents a reasonable single source of truth for advancements.

Sydr [51] was introduced to empirically measure the effect of various symbolic execution enhancements: "skiping non-symbolic instructions, AST simplification, path predicate slicing, indirect jumps resolving, and handling multi-threaded programs". Of these, only the special attention paid to handling multi-threading was novel at the time of release. To handle mutli-threading, the authors suggest implementation of a coherency model that implements "symbolic context switching" to separate the execution of each thread, but fall short of implementing a symbolic thread scheduler to explore possible data race vulnerabilities. SymCC and the following work SymQEMU [52] implement one of the most recent advancements in

symbolic execution, purely working towards speed and scalability improvements. Instead of using an offline tracing or taint system, or an IR to collect symbolic information, SymCC compiles the handling of symbolic data directly into the target binary so that it can execute with a significantly reduced performance penalty. SymQEMU does the same, but is able to execute on black-box binaries. Despite using no new techniques aside from the implementation detail, SymCC and SymQEMU achieve incredibly strong results, which indicates there is still room to make advances in the feasibility and general applicability of symbolic execution through new applications of existing methods.

Chandra et al. presented Snugglebug [53], a tool that addresses slightly different analysis goals to the exhaustive symbolic execution discussed previously. Snugglebug identifies a problem with forward symbolic execution and the prohibitive cost of executing until a deep vulnerability where the path is extremely long. This is a problem that KLEE and other prior papers address in various ways, but all continue to suffer from the exponential nature of path explosion without manual direction, despite heuristics and optimizations. Snugglebug proposes a method for *backward* symbolic execution, where a particular code location of set of locations is specified, and the tool attempts to determine reachability of the code as well as derive external inputs sufficient to reach it. Their analysis targets highly complex Java code expressing polymorphism, object oriented programming, and runtime side effects by implementing "directed call graph construction". This type of analysis, as mentioned previously, is known as *weakest precondition* analysis, to determine the over-approximated set of inputs to reach a given location in a program. This type of backward symbolic execution for weakest precondition analysis was also utilized by BAP and its derivative works.

Brucato et al. presented a method [54] for detecting input-parsing functions in programs using symbolic execution. The authors use DSE over inputs that successfully pass parsing checks as well as inputs that fail parsing checks and analyze the divergence between the symbolic traces. Once a divergence is identified, the containing function may be marked as a parser function and a set of instructions in the function is associated with the failure state. Symbolic tracing is used once again to fix-up fuzzer inputs to pass checks identified in this

set of instructions. The intuition of this approach is remarkably similar to the approach that will be explored herein, although the implementation is very different. The symbolic approach taken by Brucato relies on user intervention and the availability of known positive and negative test cases, so it is not suitable for scalable black-box analysis. The authors note that despite "encouraging results", there are significant limitations to the approach due to the use of exhaustive symbolic execution.

1.1.3 Taint Analysis

Taint Analysis is a dynamic dataflow analysis technique in which data, which may be memory, registers, external inputs, or results of system calls, are marked and tracked (*tainted*) during the execution of a program. Typically, taint analysis focuses on the *propagation* of this data after it is tainted, with an end goal of observing some or all data that is affected by the tainted data. While general static dataflow methods have existed for quite some time, as discussed in Subsection 1.1.1, dynamic data flow methods remained somewhat impractical until the industry and academic movement to C in the 1990s and the introduction of analysis frameworks that could support taint tracking, which is necessarily a very heavy-weight analysis, as it must track each "value" in the program or full system. Denning proposed [4] a model in 1976 that has evolved into this understanding of taint analysis. Denning proposes that within a program, down to the variable level of granularity, information flow can be modeled as functions of transfer from one location to another. In modern taint analysis systems, this flow is even more granular, even down to the bit level. Valgrind [55], one of the most prominent taint analysis systems still in use, was presented in 2005. It uses taint tracking over an IR lifted from machine code called VEX, and tracks tainted data propagation through a program with bit-level granularity. It also uses highly accurate representations of instructions to determine exactly what data is tainted after an operation involving a tainted data source. The authors of Valgrind utilize it in a tool called Memcheck, which uses taint analysis to determine the source of various memory errors. It is a well-known and powerful tool not only for vulnerability discovery but for development,

and can reduce debugging time significantly.

Another tool, Dytan [56] is implemented not as a binary tool but as a library that is used from the target program. Unlike Valgrind, which now provides a large API but was initially designed mostly for command-line use, it was less opinionated and provided additional formal definitions of common taint analysis terms, including propagation policy, sink and source, the authors also showed Dytan working on very large binaries such as Firefox. Valgrind also works with Firefox as a target, but it is notable that within a short time frame multiple teams saw enough value in efficient taint analysis to justify the creation of both projects. Much more recently, efforts have begun to bear results that attempt to apply taint analysis, which has generally been a theoretical static exercise or a mainly userspace exercise, to full systems. DECAF [57] introduces a method for using QEMU's TCG to analyze data taint in a full system via *Virtual Machine Introspection* (VMI). This effort required significant innovation in accuracy and speed of propagating taint at a guest instruction level.

REDQUEEN [58]'s authors primarily designed it to accomplish goals in fuzzing, discussed in the next section. However, the real innovation behind REDQUEEN was an improvement in the mechanism for performing taint analysis. The authors of REDQUEEN specifically discuss Valgrind as an effective approach to taint analysis that was far too slow for their needs. Because fuzzing is a throughput-driven task, any process which reduces throughput of fuzzing is an unacceptable tradeoff. Instead of using global or local taint tracking, REDQUEEN adopts a method that is quite intuitive: changing small parts of the input to determine whether the input *corresponds* to the state observed in the program. The authors call this technique *Input-to-State* correspondence, an apt name. In contrast, a similarly timed paper about Angora [59] details a fuzzer that does use byte-level taint tracking. However, the Angora authors intentionally trade off taint tracking accuracy for increased speed by scoping their use of taint analysis to a specific sub-problem. They use the information about data flow of tainted input to determine which components of an input to the program must be mutated to achieve additional coverage.

1.1.4 Fuzzing

Fuzz testing, or simply fuzzing, is a dynamic analysis technique where a program is run many times with many different inputs to test program robustness against unexpected external inputs. Fuzzing was initially conceived in 1962 as a method for testing the logic of COBOL programs by Saiider [60], and generated "improper relations at random". This system also accepted specifications of the data to generate, an idea that would eventually become structure-aware fuzzing. Several other researchers throughout the 1960s and 1970s investigated approaches that resemble fuzzing. Burgess [61] used the term *black-box testing* to refer to testing without visibility into the state of the software under test. Other works instead utilized static analysis of the program under test to generate test inputs for functional and robustness verification. This approach is now known as *white-box testing*. Shortly following these works, Goodenough [62] showed that several metrics, including coverage, are insufficient to fully test a program for problems. For example, they show that executing a statement inside a loop only a single time will not uncover problems with the statement occuring after many loop executions. This is intuitive, but an important idea to understand further developments in fuzzing. Many such developments seek to expand the context under which code is executed in order to discover further error conditions.

The actual term *fuzzing* was defined by Miller [63] in 1990, who pointed out that "while formal verification of a complete set of operating system utilities was too onerous a task, there was still a need for some form of more complete testing". Miller and other UNIX developers created a tool called *fuzz* that generated random data for input to system utilities. Their goal, unlike the aforementioned COBOL-centric testing, was explicitly to crash or hang programs to identify faults. This "first" fuzzer was extremely simple: it did not have any feedback mechanism, which means the testing it was used for was truly black-box. This paper was the first to demonstrate the extraordinarily wide applicability of such a simple approach, crashing "24%-33%" of programs on each system. This is an important observation about fuzzing in general. The approach is very simple, even the most complex fuzzers at their core are generating random data and attempting to overcome probability and cause

a crash. The list of observed weaknesses presented by Miller demonstrates the wide variety of weaknesses fuzzing is able to discover. They note memory errors, improper use of library functions, race conditions, and signedness bugs as common categories of weakness that *fuzz* was able to find. Miller released a follow up paper [64] in 1995, extending the use of the *fuzz* program to network and GUI applications, and introducing memory allocator faults. Their results displayed continued effectiveness of the fuzzing process. Miller and Forrester [65] were able to replicate similar fuzzing results on the Windows platform as well as on Unix.

From 2005 to the modern day at the time of writing, fuzzing is a rich research area with many sub-disciplines. Many research works focus on just one of these sub-disciplines, and others combine many ideas together. Along with advancements, the fuzzing research community arrived at a partial consensus on vocabulary used to describe the ideas employed in fuzzing research.

Generative Fuzzing

Generative or *generational* fuzzing is a test case generation mechanism where a specification for the input data is provided to a fuzzer, parts or the whole of which is modified by the fuzzer prior to each execution of the program under test. One of the first fuzzers leveraging generative fuzzing was Spike [66]. The idea behind spike was to represent network protocols, file formats, and other data representations as linear bytes in memory no matter their semantic structure. This linear structure was generated using a programming API that allowed users of the fuzzer to specify constant byte areas and variable areas of a particular data type. This is a simplistic approach, but allowed Spike to successfully pass checks that a legacy "dumb" fuzzer like *fuzz* would never pass in practice. A motivating example in Listing 3 demonstrates a use case. There is a vulnerable copy of user-provided data with a user-provided size, but the vulnerable code is hidden behind a format check of the incoming structure. A generative fuzzer then, could keep the required magic values present in all test cases while randomizing other parts of the input, including the size of the

randomized input. This allows fewer wasted executions, but requires manual intervention by an analyst to create and modify input specifications throughout the fuzzing campaign. Spike attempted to mitigate this problem by providing many formats by default along with the fuzzer, but analysis of any bespoke protocol or format would require renewed effort.

```
struct ExternalRPCLocatorStruct {
    uint16_t magic;
    uint16_t size;
    wchar_t request[]
};

bool CanDemarshalRPCLocatorStruct(ExternalRPCLocatorStruct *l) {
    if (l->magic != 0x3) {
        return false;
    }
    return CheckRequest(l->request);
}

void MSRPCLocator(ExternalRPCLocatorStruct *l) {
    struct InternalRPCLocatorStruct *i;

    if (CanDemarshalRPCLocatorStruct(l) != NULL) {
        // Vulnerable, but hidden behind a check!
        memcpy(i, l->request, l->size);
    }
}
```

Listing 3: Example of check guarding vulnerable code inspired by RPCLocator Vulnerability

Peach fuzzer [67] adopted a similar approach to Spike, but did not use specifications written in code. Instead it used an eXtensible Markup Language (XML) file containing a Domain-Specific-Language (DSL) for specifying data layouts. Peach's implementation of this DSL allows for extraordinarily well specified data models, particularly for complex file formats. In fact, Peach was later used to fuzz the Firefox browser due to its ability to generate well formed font, HTML, JavaScript, and CSS files. Despite the heavy up-front investment to specify file formats in such an intensive fashion, Peach has a more robust specification that allows repetitions, optional entries, variable lengths, and many built in data types.

Autodafe adopts an approach more similar to that of Spike, using code to specify generative formats for fuzzing. Autodafé's innovation was primarily focused on known vulnerable functions such as strcpy. Unlike Peach which simply detects crashes as its only feedback mechanism, Autodafé uses a callstack tracer that records a trace at the function-call level to determine when data from the fuzzer input testcase is used as a parameter to vulnerable functions. Detection increases the weight of that specific component of the input which is then modified by replacing it with data from an input library. The tracer is also used to provide additional context when a crash does occur.

Much more recently, Skyfire [68] addressed problems arising from generational fuzzing. The authors note that despite the advantage of generational fuzzing in passing syntactic checks, generational approaches often generate data that is semantically meaningless. This means that while generational approaches may surpass both "dumb" and mutational approaches in early code coverage as they are able to pass initial checks, these approaches still become stuck at more complex checks. A motivating example provided by the authors is a check in the eXtensible Stylesheet Language (XSLT) that an *attribute* of an *element* is valid. Generational fuzzers have little chance of passing such checks, and are reduced to the effectiveness of "dumb" or mutational fuzzers to pass them if the semantic checks occur on the unstructured sections of the generated test case's input. In addition to a grammar, or input specification, Skyfire uses a large corpus of valid example files to generate additional tests derived from the input corpus. This enhancement to generational fuzzing enabled deeper bug discovery, a key goal in modern fuzzing research. NAUTILUS [69] adopts a very similar approach, although it utilizes grammars specified using the ANTLR parser. Unlike Skyfire, NAUTILUS makes heavy use of mutation as a post-processing step to generation of well-formed input cases. Superion [70] makes additional incremental progress over NAUTILUS by emphasizing the process of "tree mutation" and "enhanced dictionary-based mutation". Dictionary-based mutation in this case refers to the random insertion of user-provided or automatically identified keywords in the input at identified token boundaries. The goal of this process is to reduce unsoundness of inputs and reduce the required amount of mutation. Superion's tree-based mutation uses the Abstract Syntax Tree (AST) of the input to recur-

sively append, remove, and replace subtrees of the AST to cause large variation to the input without relying on byte or bit mutation. Their method keeps structure and parsing validity constraints, again to reduce the number of unsound inputs that exit quickly. The authors claim significant improvements in not only number of bugs found but in the efficiency of the fuzzing process using this methodology, but note that it is not applicable to all input formats and is most well suited to highly structured input types such as XML and HTML.

Overall, generative fuzzing is a highly effective approach, especially when the target is well suited to highly structured inputs. However, the primary drawback of grammar-based and generative fuzzing has yet to be solved. Human analysts must still invest significant time into the harnessing and input grammar specification, which reduces the real-world usefulness of the approach. Software engineers are typically unenthusiastic about testing, and even less so with respect to writing specifications for input formats to their sotware.

Mutational Fuzzing

In contrast to generational fuzzing, mutational fuzzing does not utilize a specification or formal grammar describing an input file. The mutation process starts with an input seed, which can be random or empty, and modifies the input seed in random locations to exercise code paths [71]. There are infinite types of mutations, but they generally fall within a few categories. Miller [72] describes "replacing small strings with longer strings or changing length values to either very large or very small values". Bunny the Fuzzer, due to Zalewski [73] was an early mutational fuzzer that implemented bitflips, byte, word, and dword setting, deletion, replacement, insertion, and swapping of test case data. An important breakthrough from Bunny was the implementation of compiler-based coverage metrics used for feedback direction. Bunny used a crude C parser to add instrumentation to the program before compilation that allowed it to observe coverage during test case execution without expensive post-invocation tools such as gcov [74]. It used this fast coverage information to provide feedback to its mutation engine to determine which mutations were more effective, allowing

the mutation to continue trying techniques that yielded the best results.

Mutation has evolved significantly since the implementation of Bunny, and now utilizes advanced techniques such as stochastic gradient descent and random search to inform test case mutation instead of simple randomly chosen operations. Last et al. [75] proposed advancements to test case generation and mutation with two goals in mind: "generate good test cases...one that has a high probability of detecting an as-yet undiscovered error" and "prioritize test cases according to a rate of fault detection". Their novel approach uses *Genetic Algorithms* (GAs) as a search methodology for input test cases. Genetic algorithms have influenced a significant number of mutational fuzzing advancements, so it is helpful to understand the process. Zeller et al. describe the process in easy to consume terms [76]. First, test cases are modeled as chromosomes. These test case chromosomes are composed of genetic information which in fuzzing are the various components of the input, the demarcations of which the genetic algorithm may or may not be made aware of through feedback mechanisms. If initial test cases are provided, those test cases will typically be used as part of the initial population of chromosomes, but they may also be randomly generated. Once the population is initialized, the following operations proceed in a loop. First, members of the population are *selected* for mating, which involves a fitness determination to select the best individuals in the population. For fuzzing purposes, the best individuals are those whose input to the program being fuzzed generates the largest desired result. We will discuss additional feedback mechanisms further, but coverage is the most common fitness metric. The individuals selected for mating are then input to a *crossover* function which combines the input individuals to produce one or more offspring. Zeller et al. provide an example where each input is divided in half and the two combinations of halves become the offspring, but interpolation and more exotic crossover operations such as cross product are also possible. Each offspring is then *mutated* to some degree, which inserts simulated evolutionary randomness into the process. Last et al. note that the most common mutation methods are bit flips, whereupon each bit in the offspring chromosome is flipped with some probability, and uniform mutation in which the algorithm will "choose one bit randomly and change its value". Finally, the offspring are inserted into the population, and the loop continues with

selection after running the binary being fuzzed with the new population as inputs.

American Fuzzy Lop (AFL), due to Zalewski [77] is undoubtedly the most famous and most used fuzzer. It is a simple, robust mutational fuzzer that uses compiler or emulator-based coverage data as a feedback mechanism. AFL's initial release used a worklist-based method for discovering new test cases. Zalewski claims that "in contrast to more greedy genetic algorithms, this approach allows the tool to progressively explore various disjoint and possibly mutually incompatible features of the underlying data format". More specifically, if a test case discovers additional state within an execution, it is marked as interesting and added to a queue for further mutation. This process may not precisely follow the GA sequence outlined above, but it is still an implementation of an evolutionary algorithm for test case selection. AFL introduces a step not found in the above GA stages and periodically removes test cases whose coverage is a proper subset of other test cases. Its mutation steps are: "sequential bit flips with varying lengths and stepovers", "sequential addition and subtraction of small integers", and "sequential insertion of known interesting integers". Zalewski notes in a separate blog post [78] that "test case splicing" reminiscent of the previously mentioned crossover function can be used as a last resort along with block operations of duplication, deletion, and setting. Zalewski claims this set of approaches outperforms GAs, but concedes that "the mutation engine for a new fuzzer has more to do with art than science".

AFL's success made its mutation strategy the de facto approach for many papers, but work still proceeded to identify strategies to further improve the effectiveness of test case generation processes. VUzzer [79] attempts to "prioritize deep (and thus interesting) paths and deprioritize frequent (and thus uninteresting) paths". They present a motivating example similar to Listing 4 that AFL experiences a problem with. The authors point out that because AFL will discover a new path whether it takes the *true* or the *false* branch of a conditional, it may waste significant time exploring a new path "even if the path leads to an error state". In the example, time will be wasted exploring the false branch of the second conditional without further progress, because that branch will undoubtedly be discovered

first. In order to solve this problem, VUzzer proposes using taint analysis (which we discuss in Subsection 1.1.3) to track data flow along with control flow (i.e. coverage information). The authors utilize this information to inform the fitness step of their GA. The fitness operation is a function from input to fitness weight where fitness weight equals the sum of the logarithm of the block's "hit frequency" multiplied by the weight of the block. The weight of the block is equal to the number of basic blocks executed divided by the weight of the error handling blocks encountered by the input. VUzzer uses the same *crossover* function as Last of dividing inputs, or chromosomes, in half and combining them. VUzzer adds additional steps to the *mutation* of chromosomes, however. First, VUzzer inserts strings at random into the input where "tainted offsets" appear. VUzzer defines a "tainted offset" as the offset into the input data where data is read or pointed to. Next, the system modifies data in the input at offsets where data is accessed with the `lea` x86 assembly instruction. Finally, it directly modifies data where possible to change the direction of comparisons from `cmp` instructions and adds "magic bytes" where they are detected. This process reflects more closely the actual behavior of the application than simply observing, as the basic AFL implementation does, how application statistics change after running the program with a given input. However, the taint analysis and additional mutation steps slow down VUzzer significantly, so there is a large tradeoff between speed and accuracy. That tradeoff appears frequently with regards to both feedback mechanisms and mutation strategies.

Angora [59] builds further on the work done for VUzzer and implements a gradient descent search algorithm for identifying interesting inputs. The authors use a transformation from binary comparisons to constraints on input data, then utilize gradient descent to find a minimum of the given constraints. This minimum corresponds to a solution of path constraints that allows the fuzzer to move into unexplored code. A similar tool, FairFuzz [80] does not replace the mutation algorithm used by AFL. Instead, it first uses enhanced coverage information beyond that provided by AFL to recognize when branches in the program are being utilized less frequently. It pairs this additional coverage information with an enhancement to AFL's mutation algorithm. FairFuzz computes a *mutation mask* based on execution information that excludes mutations to data that causes AFL to hit common

```c
int main() {
    int sz;
    read(0, &sz, sizeof(sz));

    if (sz < 16) {
        exit(1);
    }

    char *buf = (char *)malloc(sz + 1);
    read(0, buf, sz + 1);

    if (buf[sz] == 0xd8 && buf[sz - 1] == 0xff) {
        deeper_function(buf);
    } else {
        exit(1);
    }
}
```

Listing 4: Example of a problematic check for the original AFL evolutionary fuzzer

branches. This allows FairFuzz to focus more quickly on rarely explored code paths and identify deeper program logic. In practice, these techniques primarily enhance the capability of AFL against complex string comparisons and checks against constant values. REDQUEEN [81], due to Aschermann et al. makes yet another large leap in passing complex checks on user input values. Unlike FairFuzz and VUzzer, REDQUEEN does not utilize taint tracking but is able to determine an "approximation to taint tracking and symbolic execution" by simply setting input bytes to "color" values (where there are, of course, 256 possible colors) and tracking when these input bytes arrive at comparison instructions. By testing inputs multiple times with modified "colors", REDQUEEN is able to efficiently check whether the compared data in fact corresponds to input data without requiring heavy taint tracking or emulation. REDQUEEN also implements mutations on these discovered "input-to-state correspondences": zero and sign extension, reversing, null termination, encoding, and replacement of corresponding inputs.

AFL is likely the most used fuzzer of all time, according to the AFL++ paper [82]. AFL++ is a direct enhancement to AFL that includes many optimizations and enhancements from works like FairFuzz, VUzzer, and more. AFL++ additionally supports custom mutator implementation to enhance mutator swaps for various targets. It implements "structure-aware" mutators based on work by Fioraldi [83] et al. that allows for a generational fuzzing approach embedded inside a mutational fuzzer to more efficiently fuzz complex data structures. Finally, it implements the mutation schedule due to Lyu's MOpt [84]. This mutation enhancement does not add new *operators* to AFL's mutation, but it optimizes the order in which the operators are used. The *Pilot* stage from MOpt uses probability distributions of particle swarms, which represent mutation operators, to determine how much each operator is contributing to the number of interesting cases found using it. MOpt also measures how many total attempts are made to discover an interesting case, and discovers how efficient each mutation operator is by dividing interesting cases found by number of attempts made. The *Core* stage uses the discovered optimum operator to determine which swarm is a global optimum. Finally, the *Pacemaker* mode from MOpt – which was not included in AFL++ at the time of writing but is included in the modern distribution – is able to selectively disable an inefficiently functioning deterministic stage permanently.

Included in almost all modern fuzzers is the *Havoc* stage of input mutation, which was first proposed in AFL. Wu et al. analyzed the *Havoc* approach [85] and made several improvements. Havoc selects mutation operators uniformly at random from a set of possible operators and applies them on one given seed. This generates a much more highly mutated seed and is able to overcome many roadblocks to fuzzing through its stochastic properties. The authors first found that havoc is in many cases more effective than the primary mutation option used by many fuzzers. The authors also implement a Multi-Armed-Bandit (MAB) based extension to Havoc. Mutators are still selected by the uniform distribution, however the choice of how many mutators to select is also randomized based on whether mutated test cases generated by the mutator types and counts chosen explore new program edges. The authors show that this extension to Havoc is highly effective and even beats symbolic

execution-based approaches.

From a survey of existing research, it is clear that mutational fuzzing is the preferred approach for real-world fuzzing applications. Primarily, this is due to ease of use. Generational fuzzing requires up-front time investment for any effectiveness, while mutational fuzzing can be started with zero investment and improved over time. This explains the greater interest in mutational fuzzers and the wider propagation of AFL-based fuzzers over any other fuzzer core implementation.

Hybrid fuzzing

Hybrid fuzzing refers to the combination of fuzzing with other approaches, typically either symbolic execution, taint analysis, or both. Concolic approaches such as DART and CUTE [86] (due to Godefroid and Sen, respectively) explored concrete combined with symbolic execution, known as *concolic execution*. The authors showed that in order to assist a fuzzer in exploring all paths in a binary, symbolic execution can be used to solve path constraints on executions of the program under test in order to force program execution to find new paths in the program. Sen later built on their work with CUTE to present a more capable system for concolic testing that enabled their prior work to explore coverage more quickly by reaching what the authors call "deep states". They critically observed that these deep states are not reached by a standard fuzzer without combining symbolic execution. SAGE [46], MAYHEM [8], VUzzer [79], as well as older tools such as Flayer [87] discussed above all implement some form of hybrid analysis. Driller [88] is one of the most well known examples of hybrid fuzzing. The tool uses concolic execution alongside traditional fuzzing to "alleviate path explosion". The authors specifically apply symbolic execution to fuzzing where it is most useful to pass through checks that are difficult to randomly satisfy. In the model used by Driller, programs are divided into logical "compartments" of easier-to-explore code separated by these difficult checks. Driller quickly explores the code inside compartments using a traditional fuzzer, in this case AFL, and uses symbolic execution to move between these logical compartments by generating inputs that satisfy the complex checks. QSYM [89]

uses a very similar approach to Driller, but applies DBI to execution to achieve symbolic execution instead of using emulation to apply the technique to larger real world software projects. They make several points about popular techniques. First, they do not use an IR, instead using QEMU for execution. Next, they do not utilize snapshots of symbolic executions to start from a branch and instead resume symbolic execution from entry with each new test case. QSYM also does not use a model of the external environment and allows it to be used organically. Finally, they do not focus on over-sound execution and point out that too much soundness can introduce inaccuracy and "over-constrain a path". SAVIOR [90] is conceptually very similar to QSYM but makes a large innovation in that it does not use symbolic execution as a "last resort" instead using it optimistically to help the fuzzer discover new paths. In addition, SAVIOR utilizes the UBSan sanitizer to guide the fuzzer towards *likely* bugs deriving from use of undefined behavior. SymQEMU builds on both earlier works by using a compilation-based approach inside the translation engine of QEMU to avoid needing heavyweight tracing or compiled-in symbolic execution code.

Other works also fall under "hybrid fuzzing" that use enhancements to fuzzing for other reasons. Most notable among these approaches is T-Fuzz [91]. T-Fuzz uses DSE over fuzzer inputs to detect what the authors call "Non-Critical Checks", or "NCCs" in code, which are checks that are deemed non-essential to the operation of a program, but which prevent a fuzzer from making progress. T-Fuzz identifies these checks and does not just change the direction of the comparison, but actually produces a transformed program with the branch condition inverted. The authors note that this causes a soundness problem. Any crash or bug found on a transformed program may not be present in the original binary. They mitigate this by verifying any found bugs and implementing a process for transforming a bug from a transformed program into a bug on the original program, where possible. The authors note that this fast bypassing of NCCs reduces the need for symbolic execution of many test inputs which increases throughput and exposes bugs more quickly. Other projects such as CAFA [92] were also introduced at similar times. CAFA focuses specifically on checksums. CAFA first recognizes that a checksum is being computed and then identifies the point where a comparison against a checksum occurs and inverts the direction of the branch. This allows

any input that does *not* pass the checksum check to continue execution, instead of any input that *does* pass. These tools present an extremely interesting idea that binaries need not be immutable to discover vulnerabilities and bugs, however both papers point out the severe unsoundness of such methods.

1.1.5 Prior Work Summary

The discussed work in subfields of binary application security prompts researchers to pose the question: "what is next?" Our hypothesis leads us in a direction most influenced by work in program transformation by projects such as CAFA and T-Fuzz, but with several additional goals and extensions in mind. In order to consider extending existing methods for fuzzing by program transformation, however, every category of binary analysis technique must be used. First, fuzzing must be used in order to enhance an existing (or potentially, a non-existent) corpus of inputs. Next, dynamic analysis must be used to analyze the control flow coverage of these inputs via emulation. Taint analysis must be implemented to observe the data flow along these discovered control flow paths in order to discover the effect of input data across a program, and symbolic execution and static analysis must be leveraged to perform the transformation of a program. Thus, these sub-fields cannot be viewed as disconnected and disparate disciplines but rather distinct parts of a solution to a single problem: what is the most efficient technique or set of techniques to find the most bugs in a program?

2. SYSTEM

2.1 Direction

Considering the vast field of binary analysis and fuzzing research, it is clear that there is ample opportunity to improve the process. We take inspiration from multiple prior works, notably those investigating program transformation, especially T-Fuzz [91], and those investigating methods for accessing and fuzzing "deeper logic" in programs, such as Angora [59]. The intersection of both the goals and approaches of these prior works, as well as the previously explored sub-fields discussed previously, leads to the key question: is it possible to use offline program transformation to allow fuzzing of deep branches while mitigating the performance drawbacks of online symbolic execution or taint analysis to facilitate full-throughput fuzzing? Many prior works have identified the key issue with even "smart" fuzzing with coverage-guided fuzzers such as AFL: they eventually reach a "stuck point" and are unable to make progress by discovering new inputs to increase coverage. The majority of related works tend to classify these stuck points as a single code statement, for example the check on line 5 of Listing 5.

```
1 int main() {
2     uint32_t x;
3     read(0, &x, sizeof(x));
4
5     if (x == 0x00ACAD1A) {
6         vulnerable();
7     }
8 }
```

Listing 5: Example (due to Driller) of a stuck point

Instead of considering specific `if` or `while` comparisons, we note that in general, high level programming languages encourage modular code in the form of functions (K&R note that "C programs generally consist of many small functions rather than a few big ones"). Therefore functions typically do "one thing". There are many exceptions, but by and large the recommendation is to follow this convention. We can apply this observation to fuzzing

by abstracting the idea of a stuck point on a single line of code away and focusing on what we will refer to as a "impeding function". If a function invocation behaves like a stuck point, it can be treated like one and the fuzzing process can potentially be enhanced by applying prior methods to this now-abstract impeding function. For example, consider the functions in Listing 6. There are three checks in `isok` that could be stuck points if a fuzzer was unable bypass them. However, when considering the calling context of `isok` we see that clearly, there is only one relevant check to bypass in order to reach the `vulnerable` function, the check on line 21.

```
1  int isok(uint32_t v) {
2      if (v >> 32 == 0) {
3          return 0;
4      }
5
6      if ((v >> 16) ^ 0xBEEF != 0x1337) {
7          return 0;
8      }
9
10     if ((v & 0xffff) > 0xFEED) {
11         return 0;
12     }
13
14     return 1;
15 }
16
17 int main() {
18     uint32_t x;
19     read(0, &x, sizeof(x));
20
21     if (isok(x)) {
22         vulnerable();
23     }
24
25 }
```

Listing 6: Example of an impeding function

This leads to a natural conclusion: if we model checks and assertions against the return values of impeding function such as `isok`, the assertions contained within the function can be

collapsed into the single assertion against its return value. The number of total checks that must be bypassed to fully explore a program will therefore be reduced. In fact, any number of additional assertions, each of which may cause a stuck point could be added to `isok`. Using this model these checks could *all* be ignored if the check at line 21 could be artificially bypassed. Despite the promise of this idea, there are several immediate concerns that could impede its effectiveness or validity. In general, all identified potential problems with an approach focused on a function level instead of a statement level are *soundness* problems. As works such as QSYM [93] and T-Fuzz [91] identify, a *sound* analysis in the context of fuzzing is one that does not produce false positive inputs. That is, the system does not produce a valid input for a transformed program that is invalid for the original program. For example, if the call to `isok` is replaced with a call to a function that returns `1` unconditionally, the analysis becomes *unsound* because any input will reach the `vulnerable` function, instead of the limited reaching input of the original program. We will discuss several soundness considerations taken throughout the system design of REFACE to mitigate unsoundness problems, and address the remaining soundness tradeoffs of the completed system in Subsection 2.2.5. Our hypothesis, then, boils down to:

By abstracting stuck points at a function level as impeding functions and modifying them to remove checks, deeper logic will be reached more quickly by off-the-shelf fuzzers, leading to increased coverage and bug discovery.

The first requirement towards testing this hypothesis is to create formal definitions of each critical component. Most important is a definition for impeding functions, functions that behave as if they were statement-level stuck points. Critical checks on data precede deeper logic operations using the data in most applications, but of particular interest is cryptographic decoding and decryption, compression and decompression, and media format parsing and conversion. These are all common operations present either as external dependencies or directly inside a large amount of the most-used applications for both personal computers and servers. In order to create a functional definition, we considered real-world code first. The function in Listing 7 from the SSL and Cryptography library *Wolf-*

SSL [94] demonstrates a prototypical impeding function. The result of the function call is checked and the checked value guards the deeper logic inside the `if` statement. The function `wc_RsaPrivateKeyDecode` contains many locations where an error value is returned, meaning this check is likely to fail while fuzzing with random input data.

```
1  int wolfSSL_RSA_LoadDer_ex(WOLFSSL_RSA* rsa, const unsigned char* derBuf,
2      int derSz, int opt)
3  {
4      int ret = 1;
5      word32 idx = 0;
6      /* ................ */
7
8      if (ret == 1) {
9          /* Decode private or public key data. */
10         if (opt == WOLFSSL_RSA_LOAD_PRIVATE) {
11             res = wc_RsaPrivateKeyDecode(derBuf, &idx, (RsaKey*)rsa->internal,
12                 derSz);
13         }
14         else {
15             res = wc_RsaPublicKeyDecode(derBuf, &idx, (RsaKey*)rsa->internal,
16                 derSz);
17         }
18         /* Check for error. */
19         if (res < 0) {
20             if (opt == WOLFSSL_RSA_LOAD_PRIVATE) {
21                  WOLFSSL_MSG("RsaPrivateKeyDecode failed");
22             }
23             else {
24                  WOLFSSL_MSG("RsaPublicKeyDecode failed");
25             }
26             WOLFSSL_ERROR_VERBOSE(res);
27             ret = -1;
28         }
29     }
30     if (ret == 1) {
31         /* ................ */
32     }
33     return ret;
34 }
```

Listing 7: The `wolfSSL_RSA_LoadDer_ex` function in *WolfSSL*. Line 30 is a stuck point arising from the call to `wc_RsaPrivatekeyDecode` on line 11.

2.1.1 Impeding function Criteria

Cases similar to the function above occur in other cryptographic libraries such as AXtls [95], OpenSSL [96], and more. Attempting to pass through every check in the impeding function is particularly problematic due to the number and complexity of the checks (the `wc_RsaPrivateKeyDecode` function specifically has 17 return locations that prevent passing the stuck point check). If the idea we identified prior could be applied, however, the checks inside the function could be effectively collapsed to a single check, a large reduction in the amount of data that must be correct for fuzzing to progress. This leads to the first observation toward a formal definition: the functions we identify should be functions that operate on data from external input sources. If the data operated on is not influenced by external input, there is no feasible method for checking or ensuring soundness. Furthermore, reaching further deep logic may be possible by modifying the control flow of sections operating on non-external data, but reaching deeper logic is unlikely to yield bug discoveries if the data passed to the deeper logic code is not controlled by the fuzzer. The first *criteria* can be expressed as:

C1: Impeding functions receive external input as parameters.

In order for a function invocation to affect control flow in a similar way to a statement-level stuck point, the return value of the function or data set by the function must be checked against some other value. In order to implement an analysis tool capable of recognizing these checks and synthesizing data capable of satisfying them, these values must be constant or otherwise available via static analysis. Otherwise, over a single or small number of executions of the program, there is no effective method for determining whether a value used in a comparison is a constant. For example, a CRC32 value could be used to avoid a Time-of-check-to-time-of-use (TOCTTOU) bug by ensuring data used at a later point in a program is the same as data used earlier in the program. A function performing this check would be an inappropriate function to modify because it would be difficult to generally analyze the

soundness of operations against the return value of the modified function. In addition, functions must have multiple possible return values. If a function only returns `1` for example, even disregarding the impossibility of its return value affecting control flow, the function is not possible to modify in a way that enables bypassing checks. Not only must a valid function have multiple possible return values, but during execution it should exhibit an *unbalanced* distribution of return values. An example is a function that returns `true` on success or `false` on failure. If the function performs complex checks against the external input it receives as a parameter, it is likely to return one value (in this case, `false`) significantly more often during fuzzing. It is sometimes, but not generally, possible to determine which of a set of returned values indicate an error condition and which value may allow execution to proceed into deeper code. If a function is observed to return each possible return value with a reasonably uniform distribution, it is unlikely to cause the fuzzer to become stuck. Finally, in order to create a stuck point, the function's return value must modify control flow in some way. We can define additional criteria based on these observations:

C2: Impeding functions must return multiple possible values.

C3: The distribution of the values returned by an impeding function must be skewed.

C4: The return value of an impeding function must affect the control flow of the program.

With the four criteria identified above, a significant number of impeding functions can be identified that may present a problem to fuzzers, but which could feasibly be modified to allow fuzzing to proceed down previously inaccessible paths. However, many functions satisfy the aforementioned three criteria but may be very difficult to modify in order to avoid causing severe unsoundness. As discussed previously, unsoundness, where inputs are discovered that are not reproducible with the original program, is acceptable and will be addressed later. However, a program may be modified to the point where it does not exe-

cute correctly at all, or executes so incorrectly that no valuable fuzzing can be done on the modified program. A large portion of these problems can be averted by avoiding functions with certain side effects. Specifically, writes to non-parameter data locations are problematic, as are system calls, especially system calls whose parameters are non-constant. To preserve maximum soundness, all operations performed by a function except for the stuck point checks it performs must be replicated in a replacement function. This lends itself to the fourth and final criteria for impeding functions:

C5: Impeding functions must be mostly side effect free, and their extraneous functionality must be replayable.

The definitions of "replayable" as well as "side effect" are left intentionally vague. In the general case, it is not possible to determine whether the essential functionality of a function can be preserved while removing non-essential checks. It is, in fact, not possible to determine whether a check is essential or not. T-Fuzz, as discussed earlier, refers to checks as either NCCs or CCs depending on whether they are "non-critical" or "critical". We must follow a similar methodology and make a best-effort approach. Ultimately, the tradeoff becomes one of runtime versus accuracy: a wholly unsound modification is likely to exhibit highly unsatisfactory behavior (anecdotally, mostly crashing) when run with inputs that do not cause problems for the original binary, despite not reaching deeper logic. We will return to this idea of a post-processing step for validation purposes in Subsection 2.2.5. Milewski [97] asserts that "pure" functions exhibit the following characteristics:

1. "Returns the same result every time it's called with the same set of arguments. In other words a function has no state, nor can it access any external state."
2. "A function has no side effects. Calling a function once is the same as calling it twice and discarding the result of the first call."

This definition is very strong, and many C functions (including those previously discussed) violate them. Therefore, for the purposes of this work we make some adjustments.

First, we remove the requirement for the "same set of arguments". Machine code is rife with pointers, and a different instance of the same type of object poses no immediate problem for analysis. Next, we hypothesize that the real inhibitor arising from external state is *mutation* of that state, not access to it. Therefore, a modification of external state is prohibited but read-only access to it is not, except for access to data external to the program, for example via a `recv` system call. Happily, this modified definition can be summarized by simply taking the second criteria presented by Milewski, using our newly modified definition of a "side effect". These criteria narrow the definition of an impeding function considerably, mostly allowing categorization of relevant functions that may present effective targets for modification. However, none of the criteria present a method for actually identifying these impeding functions in binary code, synthesizing maximally sound modifications, or applying and verifying these modifications. Before addressing these engineering challenges, several questions must first be answered.

2.2 Research Questions

2.2.1 RQ1: How common are impeding functions in real-world code?

Despite the appealing hypothesis, there is little value in searching for, analyzing, and modifying functions that rarely appear in the wild. While verifying the exact assertions discussed later over a large corpus of public code would be an intractably large task, we make a general assertion that many functions satisfying these criteria are found in similar types of code. Generally, with many exceptions, code that performs similar functions can be found inside functions with similar names. The function *WolfSSL* discussed above, `wc_RsaPublicKeyDecode` gives a starting point: decoding functions are an obvious first target. We manually analyzed popular open source libraries and identified common identifiers and keywords that may signal a function that is likely to be a valid impeding function.

The methodology for this search was as follows. First, we downloaded 1000 packages listing C as a primary language from the Debian [98] repository and extracted. Then, each C source file and header was parsed using a syntax-aware parser and each function identifier

Table 2.1. Table of keywords and frequency of appearance in 1000 samples of C code from the Debian sources repository containing 107071 functions.

Keyword	Count	Avg. Rel. Frequency (names)	Avg. Rel. Frequency (calls)
`decode`	228	2.41%	0.83%
`decrypt`	57	0.35%	0.06%
`extract`	177	1.31%	0.37%
`decompress`	50	0.49%	0.11%
`deserialize`	20	0.04%	0.01%
`inflate`	22	0.21%	0.11%

was recorded. Additionally, the number of *locations* where a function containing each keyword was invoked was recorded. The first result is the raw number of appearances. For each package, each identifier was checked case-insensitively to *contain* each search term. If any function identifier contained a given search term, the "number of appearances" was increased by 1 for that search term. This statistic gives an insight into how common functions with these keywords are across many different types of code. Next, for each package containing at least one occurrence of a given keyword, the relative frequency of the identifier was calculated by proportion of function identifiers containing the keyword. Finally, the proportion of function invocations invoking the function whose identifier contains the keyword was calculated. For example, consider a package with 100 functions total that contains one function whose identifier contains the keyword `decode` and which has 1000 total function invocations in its source code. That package would have an appearance count of 1, a relative frequency (by function) of 1%, and a relative frequency (by calls) of .1%.

These results make a strong case for the wide applicability of our approach. In total, 554 out of 1000 packages analyzed contain at least one instance of a function matching the keywords, meaning over half of the software may contain an impeding function. It should be noted that this approach for surveying packages creates an under-estimate, as many actual impeding functions have different naming conventions, or may not contain a keyword in their name at all. However, this approach gives an intuitive result that there are likely many functions that fit the criteria. In addition, functions containing "decode" average 1% of all function call targets throughout the 1000 packages surveyed. Not only does this indicate

these functions are common, it also indicates that these functions are *used* commonly, and often multiple times in the same program.

2.2.2 RQ2: Can impeding functions be identified in real-world binary code?

To identify whether a given function is an impeding function, we must effectively evaluate whether it satisfies the criteria presented above. First, we must decide how to determine that a function receives external input. Static methods, including static dataflow analysis, are possible. However, static methods are slow and as discussed in Subsection 1.1.1 are imprecise. This means, particularly in this use case, static analysis may often produce a *false negative* result and conclude that a function does not receive external input when in fact it does. Symbolic methods are also possible, using a framework such as `angr` [99]. However, symbolic methods have several key weaknesses. First, symbolic execution is slow due to multiple factors. Chief among them is the overhead of constraint solving, which increases with the depth of an analysis. We want to reach deep locations in code, so this is a significant concern. Second, pure symbolic execution will waste time. The intent of this work is to reduce impedance to fuzzing throughput by removing roadblocks. Therefore, pure symbolic exploration of the full state space of a binary is unnecessary. DSE could present a solution by utilizing symbolic analysis over a concrete execution of the program. In fact, a prior implementation of REFACE utilized dynamic symbolic execution for this purpose. However, we discovered that even DSE was not performant enough and created unnecessary overhead. The logical conclusion, then, was to use dynamic taint analysis. Dynamic taint anlysis is a more scalable approach than symbolic execution as it does not require the overhead of constraint solving, nor does it require the memory overhead of maintaining multiple disparate states during execution or exploration. In addition, taint analysis is perfectly suited to the problem of tracking data from external input through program execution and discovering locations containing or modified by that external data, even through copies and transformations.

To identify impeding functions in binary code, we begin with the original goal of enhancing the effectiveness of fuzzing. Fuzzing, as discussed previously, utilizes input mutation or generation to test a program via its external inputs. Therefore, if we begin with the design decision that REFACE will be integrated into a pipeline along with a fuzzer, it becomes obvious that the inputs the fuzzer uses to perform each test of the program can be utilized as inputs in taint analysis as well. This serves two objectives. First, it enables the use of taint analysis over an input-agnostic method such as symbolic execution. Second, it allows REFACE to operate using a view of the fuzzer's current state. For example, if the fuzzer is currently stuck attempting to pass a particular check, it will not have generated any inputs that pass through that check. It guaranteed to, however, have generated inputs that *reach* the check in question. Those inputs will be used as input to REFACE along with the binary program under test. By analyzing the program's behavior under the fuzzer's current set of inputs (called a *queue* by AFL++), we can begin to reason about relevant function behavior. This reasoning is broken down into multiple stages, each implementing a narrowing constraint on the stage before it and proceeding until finally only the set of functions in the binary that satisfy the criteria remain.

Stage 1: Stuck point identification

Before functions themselves can be analyzed, we must build an understanding of where the fuzzer is becoming impeded. Logically, a stuck point is a single location in code where the fuzzer is not able to progress. In binary code terms, it is a conditional control flow transfer where some target is either never taken or taken very rarely. For example, in the code discussed previously in Listing 7, the `if` statement on line 30 compiles to assembly similar to that in Listing 8. This code is a complex example, as it has two layers of checks leading to the *actual* stuck point of concern on line 16. However, the check and jump at line 8 is also a stuck point, where the second depends on the first. The first stage will identify both of these locations as stuck points.

```
_:
    mov rbx, 1                  # ret is initialized to 1

    # ......................... # Code prior to call omitted for brevity

    call wc_RsaPrivateKeyDecode
    cmp rax, 0                  # Compare the return value against 0
    jge no_wolfssl_error        # If rax >= 0, no error occured
    mov rbx, -1                 # ret is set to -1 if an error occurred

no_wolfssl_error:
    cmp rbx, 1
    jne exit_and_return         # If ret is not equal to 1, no deeper code is reached

    #.......................... # Deeper code omitted for brevity

exit_and_return:
    mov rax, rbx
    ret                         # the function returns the value of ret
```

Listing 8: Example assembly code of motivating function (some non-relevant assembly removed for brevity).

To identify stuck points, the first stage uses a tracing program without taint analysis to obtain a simple execution trace of the program for every input in the current fuzzing queue. Algorithm 1 presents a formal algorithm for this tracing stage. The inputs to the algorithm are a trace of program counter addresses and the opcode of the instruction at each instruction executed, and a cutoff value between 0 and 1 the ratio between "next" and "target" edges of a branch indicating it as a stuck point. The algorithm proceeds by iterating over each trace entry, recording the last comparison instruction seen. In the amd64 architecture, comparisons, or the actual location of the stuck point do not occur at the same location as the associated control flow transfer. When a conditional control flow transfer is reached, for example the `jge` instruction discussed above, it is entered into a mapping of branch locations to their corresponding comparison instruction and the next and target instructions that form the two outgoing edges from the containing basic block. In addition, hit counters for the next and target edges are created. Finally, when a trace entry has a program counter

corresponding to a next or target hit counter, the hit counter is incremented. After all trace entries have been checked, each stuck point's hit counters are checked to determine if the edges were traversed an imbalanced number of times, and a set of stuck points meeting this criteria are returned.

The algorithm is parameterized on the RA ratio because it allows for greater flexibility. Some campaigns may benefit from modifying even less difficult-to-satisfy impeding functions. For our campaign, this value is set to 0.9 which indicates that one edge of a branch is taken at least 90% of the time, while the other is taken 10% or less of the time. In general, the greatest indicator that a location may be a stuck point is a complete imbalance of edge traversal. That is, one edge is taken 100% of the time. A known violation of the assumption that such a comparison presents a stuck point is the comparison at the head of a loop. False-positive stuck points deriving from loop conditionals can be easily filtered out at a later stage using separate loop detection logic leveraging static analysis, or for a coarser approach, all stuck points with one edge having a single traversal and the other edge having many can be filtered.

The trace used as input to Algorithm 1 is obtained using the QEMU userspace emulator. At each execution of a translated instruction, the program counter is reported by the emulator to a consumer program. The consumer program maintains a copy of the binary and translates the program counter to an offset in the binary program and disassembles an instruction at the translated offset. The augmented program counter, translated instruction trace is then passed into the the code implementing Algorithm 1 and the final result of the first stage's analysis is returned. The returned set specifies each comparison operation in the binary found during execution of the binary program with a single input that exhibits stuck point criteria. This process repeats for each input in the fuzzing queue, and the set of all stuck points is used as input to future stages.

Algorithm 1 Stage 1 Trace

Inputs:

TR, an amd64 execution trace of $\langle o, p \rangle$ opcode, program counter pairs
RA, a cutoff ratio qualifying a branch as imbalanced
C, a set of opcodes for comparison instructions that set flags
J, a set of opcodes for conditional control flow transfer instructions

1: **procedure** TraceBranches(TR, C, J)
2: $S, N, T, l \leftarrow \emptyset, \emptyset, \emptyset, \emptyset$
3: **for all** $\langle o, p \rangle \in TR$ **do**
4: **if** $o \in C$ **then**
5: $l \leftarrow p$
6: **else if** $o \in J \wedge l \neq \emptyset \wedge p \notin S \wedge p \notin N \wedge p \notin T$ **then**
7: $n \leftarrow p +$ `len(o)`
8: $t \leftarrow$ `imm(o)`
9: $S[p] \leftarrow \langle l, n, t \rangle$
10: $N[p] \leftarrow 0$
11: $T[p] \leftarrow 0$
12: **else if** $p \in N$ **then**
13: $N[p] + +$
14: **else if** $p \in T$ **then**
15: $T[p] + +$
16: **end if**
17: **end for**
18: **return** S
19: **end procedure**

20: **procedure** ChooseStuckPoints(TR, RA, C, J)
21: $R \leftarrow \emptyset$
22: $S \leftarrow$ TraceBranches(TR, C, J)
23: **for all** $p \in S$ **do**
24: $\langle s, n, t \rangle = S[p]$
25: $nh \leftarrow N[p]$
26: $th \leftarrow T[p]$
27: **if** $|nh - th| > RA \cdot nh + th$ **then**
28: $R = R \cup s$
29: **end if**
30: **end for**
31: **return** R
32: **end procedure**

Stage 2: Identifying input-consuming functions

By identifying all stuck points, we obtain an understanding of where the fuzzer encounters problems proceeding. However, more analysis must be done to discover which functions cause these stuck points to occur. The first step to analyze functions is to determine the set of functions across the current fuzzer input queue that receive user input. To accomplish this, we leverage dynamic taint analysis to track data from exernal inputs like reads from standard input or network sockets through the program. To facilitate performant taint tracking, we used `libdft64` [59], a project created by the authors of the Angora fuzzing project. `libdft64` in turn utilizes Intel's PinTool [100] to obtain information about the instructions underlying a dynamic program execution. PinTool provides callbacks to user code upon certain processor events, for example on each instruction executed or before and after a system call occurs. `libdft64` uses several of these callbacks internally to *propagate* taint across data at runtime.

In the case of our implementation of this stage, we enable taint tracking of all data from external sources using file descriptors. This includes data read from the `read` and `pread64` system calls, which allows for flexible taint tracking of data from `stdin`, files, and network sockets. For the prototype implementation of REFACE, only programs receiving input via files and `stdin` are considered to reduce scope. When data enters the system, `libdft64` tags it internally for tracking. At any point the tagged data is accessed, a callback to `libdft64` is triggered to *propagate* taint by tagging locations the tagged data is copied into. Various taint analysis systems propagate taint differently, and `libdft64` only propagates *explicit* dataflows. This can be demonstrated with the sample program in Listing 9. In this example, data comes into the system and the return value is set to 1 if any byte in the input was zero, and zero is returned otherwise. `libdft64`'s hooks will cause `externaldata` to become tainted after the `read` system call completes. From there, the data flow during the `memcpy` on line 7 is an *explicit* flow, which `libdft` tracks. Therefore, the full contents of `internaldata` will become tainted after the `memcpy`. The data flow from the constant 1 into the variable `rv`, however, is an *implicit* data flow. Whether the data flow occurs or not has a dependency

on the value of tainted data (in this case `internaldata`) but is not directly copied from that tainted data.

```
int main() {
    int rv = 0;
    uint32_t externaldata;
    uint32_t internaldata;
    read(0, &externaldata, sizeof(externaldata));

    memcpy(&internaldata, &externaldata, sizeof(internaldata));

    for (int i = 0; i < sizeof(internaldata); i++) {
        if ((internaldata >> (i * 8)) & 0xff == 0x00) {
            rv = 1;
        }
    }
}
```

Listing 9: Example of taint tracking via explicit and implicit flows.

In addition to not supporting implicit flows, `libdft64` also tracks tainted data at the byte level instead of the bit level as some other taint analysis tools are able to do in exchange for additional speed and ease of use. Despite these two limitations, it allows full tracking of data from source to sink which facilitates all taint analysis dependent stages of REFACE. In order to identify input-consuming functions, external data is marked as discussed above using hooks on system calls using file descriptors. Each time a function invocation occurs, a callback is triggered that begins a scan of each argument passed to the function. The number of function arguments is identified offline using static analysis by the `angr` binary analysis framework. The algorithm for scanning a function argument is given in Algorithm 2.

The first component of Algorithm 2 explores in a breadth-first fashion through the memory that may be pointed to by a function parameter. This function parameter is inserted into a new tree structure as the root. At each step, it collects the child items of the pointer it is currently considering. Nodes in the tree may be either pointers or data nodes, but

Algorithm 2 Stage 2 Function Argument Scan

Inputs:

V, The value of an argument register

```
1: procedure ArgumentScan(V)
2:     Q = NewQueue()
3:     T = NewTree(Q)
4:     push(Q, NewPointerNode(V, ∅))
5:     while Q not empty do
6:         P = front(Q)
7:         for all C ∈ CollectChildren(P) do
8:             InsertAndMerge(C, T)
9:             if isptr(C) then
10:                push(Q, C)
11:            end if
12:            addChild(P, C)
13:        end for
14:        if children(P) = ∅ then
15:            removeFromParent(P)
16:        end if
17:    end while
18: end procedure
```

data nodes may only be leaves of the tree and represent data that is tainted. Pointer nodes may be any interior node of the tree and cannot be leaves, and can have multiple children. Effectively, a "pointer" node represents a region of memory that lies behind that pointer, up to the parameterized walk size limit. After collecting the child entries in the memory pointed to by the current node, newly discovered notes are inserted and merged into the tree. The merge operation handles a case where a newly discovered node, or multiple newly discovered nodes, either overlap each other or the existing data nodes in the tree. If this occurs, adjacent and overlapping nodes are merged together into a single node holding the full data and extents of the union of all overlapping and adjacent nodes. It also handles the case where a pointer node is discovered that is already present in the tree. If this happens, a cycle would occur if the node was inserted into the tree, so such nodes are simply deleted without insertion. Finally, if a pointer node is considered but has no discovered child entries in the tree, it is removed from the tree to preserve the invariant that pointer nodes cannot

Algorithm 3 Stage 2 Child Scan

Inputs:

P, The possible pointer to tainted data

SS, WS, the parameterized stride (pointer) size and walk size

```
1: procedure CollectChildren(P, SS, WS)
2:     b ← copy(P[:WS])
3:     C, DR, o ← ∅, ∅, ∅
4:     for i up to len(b)/SS do
5:         if mapped(b[i]) then
6:             C = C ∪ NewPointerNode(b[i], P)
7:             o ← i
8:         else
9:             for j = 0 up to SS do
10:                if tainted(P + (SS ·i) + j) then
11:                    if o ∉ DR then
12:                        o ← i
13:                        DR[o] ← 1
14:                    else
15:                        DR[o] + +
16:                    end if
17:                else
18:                    o ← i
19:                end if
20:            end for
21:        end if
22:    end for
23:    for all d ∈ DR do
24:        C = C ∪ NewDataNode(P + DR[d], copy(b[d:DR[d]))
25:    end for
26:    return C
27: end procedure
```

form the leaves of the tree.

The inner function of the algorithm, shown in Algorithm 3 performs a 1-dimensional walk over the data pointed to by the argument. It proceeds in steps of pointer size, checking if each pointer-sized value is a valid mapped pointer. If it is, it creates a new child node and adds it to its list of collected child nodes. If a value is not a valid mapped pointer, each byte is iterated over and checked for taint. If the byte is tainted, its offset is either added to an

existing range of tainted data or a new data range is created beginning at its offset. After each disjoint range of tainted data is identified, new nodes are created for each data range. Finally, the set of child nodes is returned to the caller.

These trees representing the tainted memory are captured at the entry of a function. During the execution of the function, memory read operations are instrumented and any read addresses are tracked. At the return from a function, the corresponding tree is examined and any data from which no memory reads occurred is discarded from the tree. This solves a problem where the recursive memory walk examines data that is not actually input data to a function. For example, a function may receive a pointer to a stack buffer of length 80 containing data from an external interface that is located directly before an integer value that also contains external data. While walking the pointer passed as input, the integer value will be treated as continuous tainted data from the buffer by the scan algorithm. However, if the function is well behaved and receives only a pointer to the input buffer, it will never read data from the integer and will allow it to be excluded from the tainted data tree.

Functions may be invoked multiple times during a program execution, and each invocation must be checked. That is, a scan cannot be cached per function, nor even per callsite. This is due to the aliasing problem, discussed in Subsection 1.1.1, and which here allows a function to be called with the "same pointer" that may have been reassigned from some previously untainted data to point to newly tainted data. Not analyzing a function on subsequent invocations would cause possible false negative results. Therefore, this process implements a "shadow callstack" that is pushed to on function invocation and popped from on function return. As an optimization and scope limitation, only programs within the main binary program under test are tracked. This process proceeds until program exit, where the set of functions that received any user data is returned, or in simple terms the list of functions verifying C1.

Stage 3: Return values causing stuck points

Once functions receiving user input have been identified, the set of functions must be narrowed down to determine which of them actually cause a stuck point to occur. Specifically, we must determine which functions' return values are found in one of the arguments to a comparison instruction that is associated with an imbalanced branch point. This analysis can also be performed with the help of taint analysis utilizing `libdft64`. As inputs, this stage of analysis takes the list of addresses where stuck point comparisons occur and the list of functions identified during the previous stage as receiving external input. Upon returning from a function in the list, the return value is tainted. Then, until the next call or return instruction, each instruction address is checked against the list of addresses of stuck point comparisons. If there is a match, the operands of the instruction are checked for tainted data. If any is found, the function is added to the returned list of functions. Functions passing this narrowing filter exhibit C4, and therefore because they have now been identified to cause a stuck point, which are defined by Stage 1 to posess a skewed distribution, they must also exhibit C3 and C2. In our analysis, C2 is also verified using static analysis.

2.2.3 RQ3: How should impeding functions be modified?

To this point, we have discussed at length what functions should be considered for modification to test the hypothesis that modifications can be applied to at a function level to enhance throughput of a fuzzer. At this stage in analysis, these functions have been identified with high accuracy, but no assessment has been made as to their actual fitness for modification. As an example, consider the program in Listing 10.

This program exhibits a poor candidate for patching. Assuming it returns more than one possible return code, function `run` exhibits C1-4. However, it is a poor candidate because it likely encapsulates nearly all of the program's functionality. This function could be modified, and doing so would likely enable the fuzzer to obtain coverage of line 13 as well as line 11. However, it would also eliminate any deeper logic from being reached if the

```c
int run(char *input_data) {
    /* ..................... */
}

int main(int argc, char **argv) {
    char *data = read_file(argv[1]);
    int success = run(data);


    if (!success) {
        printf("Failure!\n");
    } else {
        printf("Success!\n");
    }
    return success;
}
```

Listing 10: Example of a poor impeding function patch candidate.

deeper logic is contained as subroutines of the `run` function. This type of function presents a difficult challenge. How can we detect when a function is a good candidate for patching? There are several heuristics that can be applied to filter out simple cases. Call depth could be examined from main, and a cutoff can be used to eliminate "main-like" functions such as the `run` function above. In a similar fashion, functions whose own call trees are shallower could be prioritized. For example, the `run` function above could make many subroutine calls, but a base64 decoding function is unlikely to make many. However, it is easy to find counterexamples to many heuristics. Because of this, we elect to perform no filtering to attempt to evaluate fitness for modification. Patch synthesis is reasonably low overhead, requiring a limited number of dynamic taint analysis run per binary, and it is significantly easier to evaluate the effectiveness and performance of a modification *after* it has been applied rather than than attempt to determine whether an as-yet-applied patch will function well.

Stage 4: Patch synthesis

After impeding functions have been identified by stages 1 through 3, modifications to each impeding function must be synthesized using a combination of statically and dynamically recovered information about the function. We will refer to binaries generated by the process of synthesizing and applying patches to the original binary under test as variant binaries. First, we patch only one impeding function per output binary. Put differently, for each identified call to an impeding function identified by previous stages, one output binary will be generated, each with a different patch applied. There are several reasons to only modify a single impeding function instead of alternate approaches, such as modifying all or some other subset. Chief among them is the goal of preserving soundness if possible. Any number of impeding functions may be identified, and while it is difficult to maintain soundness of newly discovered inputs for a variant binary when re-applied to an original binary, it is orders of magnitude more difficult when multiple functions have been replaced with synthesized versions. This is a scope-reducing decision, however it is also a scalability consideration. Given a production-ready implementation of this research work, the desired workflow would likely proceed as follows:

1. A fuzzing job or fuzzing jobs are created
2. The fuzzer eventually becomes stuck
3. REFACE is run to generate a set of variant binaries
4. A smaller-scale fuzzing job is created for each variant binary and run for a limited period of time
5. Inputs from variant binary fuzzing jobs that proceed past the stuck point in the original binary are re-hosted to the original binary (if possible)
6. Other variant jobs are stopped and normal fuzzing proceeds until the fuzzer becomes stuck again, and the process begins again from step 2

If the objective of fuzzing variant binaries was simply to find additional bugs in the variant binaries, creating a variant with multiple patches may be preferable. The probability of unsoundness leading to a crash instead of simply incorrect behavior compounds with the addition of further patches. However, because the objective is to find bugs in the *original* binary, a balance between allowing the fuzzer to proceed unimpeded and preserving as much of the original code as possible is desirable. In addition, we only apply a patch to a particular *callsite* as mentioned above, instead of either replacing all calls to the function with calls to the replacement or replacing the actual body of the function with modified code. This decision is once again motivated by reducing unsoundness, but also serves to narrow the focus of a given impeding function modification on passing through the associated stuck point. Consider a function such as a base64 decoding function. If the function is called in two separate locations in the binary, one of which decodes constant data compiled into the program and the other which decodes externally-supplied data , it serves no purpose to modify the function when called to decode constant data. In fact, this could *only* cause unsound behavior, and is thus avoided. When a patch is applied, it is applied to a specific callsite of a specific impeding function, and only that patch is applied to that variant. We have now considered when patches will be applied, and must turn our attention to the method for synthesizing these patches. Desirable properties for a modified function are as follows.

Modifying return value distribution

First, the impeding function should be modified to return the same possible return values as the original function, but with an inverse or otherwise modified distribution. In this implementation, we hand control over the return value from the variant function directly to the fuzzer. Based on prior work on AFL [77] and other fuzzing advancements, the fuzzer will receive feedback based on the value chosen for that component of the input data, and will observe different coverage of the program. In order to identify the return values from the impeding function, static analysis is leveraged using the `angr` [99] framework. The algorithm

Algorithm 4 Stage 4 Return value identification

Inputs:

A, the address of the impeding function

```
1: procedure IdentifyReturnValues(A)
2:     R ← ∅
3:     O ← returnLocations(A)
4:     for all c ∈ callers(A) do
5:         for all o ∈ O do
6:             D ← reachingDefinitions(functionAt(A), rax, [c], o)
7:             for all d ∈ D do
8:                 R = R ∪ AllConcreteSolutions(d)
9:             end for
10:        end for
11:    end for
12:    return R
13: end procedure
```

for identifying the return values for a given function is given in Algorithm 4.

Because return values are returned (in our case) in the `rax` register, a naive approach that may work well on source code of observing locations where a `return CONSTANT` statement occur is impossible. Instead, a Reaching Definitions analysis, mentioned briefly in Subsection 1.1.1 must be utilized to determine the locations in code where the value of the `rax` register are *defined*, or set, without being *killed*, or overwritten with another definition, before a return statement occurs. This complex analysis is abstracted away here, as it is handled by the existing `angr` framework. However, these reaching definitions are symbolic. Constant values defined inside the function will be easily identified and the symbolic definition can be *concretized* to obtain a real value or set of values. However, concrete values passed into the function and then returned will not be discovered. To solve this, we pass the *execution context* in this case an artificial static definition of a call stack. This call stack can be arbitrarily large, but here we limit the context to a single outer function, as benefits are diminishing past this point. Then, for each artificial callstack possible for the function under test, observation point at each return location, all concrete solutions for each definition are

added to the set of constant return values, and the full set is returned.

Once the set of constant return values is identified, the return value component of the patch can be fully synthesized. First, an array of the constant return values is created. Then, code to consume a value of appropriate size to index into the array directly from the fuzzer is added, and finally, code to return the value in the array at the indexed location is added as the final statement in the patch. Using the motivating example function from *WolfSSL* (Listing 7) as an example, the return component of a patch might resemble the code in Listing 11.

```
int patched_wc_RsaPrivateKeyDecode(const byte* input, word32* inOutIdx, RsaKey* key,
                        word32 inSz) {
    /* ..................... */
    const int return_values[] = {
        -110, // MP_INIT_E
        -132, // BUFFER_E
        -140, // ASN_PARSE_E
        -142, // ASN_GETINT_E
        -146, // ASN_EXPECT_0_E
        0,    // No error
    };
    uint8_t idx = ConsumeUint8FromFuzzer();
    return return_values[idx];
}
```

Listing 11: An example of a return value re-distribution patch.

More complex return paradigms that "return" data to the caller by setting the value pointed to by an argument or by setting a global value are more error prone and are unsupported by the prototype implementation of REFACE. However, this is not an insurmountable challenge, and this possibility will be discussed further in Chapter 4.

Replicating memory behavior

In addition to modifying the return value of the function, the behavior of the function with respect to memory reads and writes must be replicated as closely as possible to avoid unsound behavior. This is implemented via analyses based on dynamic taint analysis. First, as discussed in Subsection 2.2.1, many impeding functions also copy tainted data to other locations, often from some type of input buffer to an output buffer. This data copying presents an opportunity to directly pass fuzzer data to deeper logic, as the data is known to be accessible via external inputs. In order to provide fuzzer data as output, the location of this output must first be identified. The algorithm from Subsection 2.2.2 is used on entry to the impeding function during dynamic taint analysis to take a "snapshot" of tainted memory. Unlike the previous use of this algorithm, however, a *second* snapshot is taken at the exit of the function. Then, the two tree structures are compared to find the set difference of tainted memory from the exit snapshot from the entry snapshot. This gives the location in memory of the output data, which is then used to synthesize the part of the patch that will consume fuzzer data and write it directly to the output location(s) identified. In order to avoid copying too much data, the sizes of the various regions of data are observed across dynamic symbolic execution over each input in the fuzzer queue. For each tainted location in the calculated set difference, the maximum and minimum sizes are recorded and used to parameterize the data consume functions from the fuzzer. For a typical `base64_decode` function, the patch after applying memory behavior replication and return value re-distribution, the patched function may resemble the example in Listing 12.

Memory behavior modification in the prototype implementation of REFACE is limited to copying of tainted data with the simplifying assumption that only functions correctly verifying criteria C5 are of interest. Additional memory operations could be replicated by performing additional dynamic or symbolic analysis, but function summarization and translation is out of scope of this project. Additional discussion in this area is covered later in Chapter 4.

```
int patched_base64_decode(char *outb, char *inb, size_t size) {
    size_t dataSize = ConsumeDataInRangeFromFuzzer(MIN_SIZE_SEEN, MAX_SIZE_SEEN);
    ConsumeBytesFromFuzzer(outb, dataSize);
    const int return_values[] = {
        0, // Success
        1, // Failure
    };
    uint8_t idx = ConsumeUint8FromFuzzer();
    return return_values[idx];
}
```

Listing 12: An example of a memory behavior and return value re-distribution patch.

2.2.4 RQ4: How can modifications, once decided upon, be applied to a binary?

Using the above analyses, patches can be synthesized that implement the fuzzer-relevant operations of an impeding function. Applying these patches, however, is another challenge. Stage 4 synthesizes patches as C code in order to simplify memory accesses relative to argument values. However, transforming C code into usable machine code, inserting the machine code into the binary, and restructuring control flow to use the new code instead of the original function are all non-trivial tasks. Several projects exist to perform each of these operations separately, but all either were insufficient on engineering grounds, either containing bugs or not supporting the target architecture amd64, or were not able to correctly apply the generated patches to the binary to generate a variant binary.

The first task of patch application is compilation from the C code patch synthesized in stage 4 to machine code that can be inserted into an existing binary. To perform this operation, we created an LLVM pass capable of taking nearly-arbitrary C code and outputting a single function of machine code using LLVM inlining called Squishy. This pass was inspired by a prior project, SheLLVM [101] that implemented a "shellcode compiler" using a similar LLVM pass. Unfortunately, the SheLLVM project is not maintained, and several steps of the pass it implemented resulted in incompletely inlined code. Squishy proceeds in several steps. First, functions are recursively inlined into their callers until only a single function (main)

remains, then removing newly dead code as a result of the inlining. Next, global variables are inlined into the beginning of the main function and transformed into stack variables so they can be made position independent. Finally, any newly undefined calls are removed and the resulting LLVM module is verified and compiled.

The patch is applied by using the LIEF [102] binary parsing and modification library. A new loadable segment is added to the binary and the newly compiled patch is added to the segment. Then, the callsite of the original impeding function is modified such that the call transfers control flow to the newly compiled patch function instead of the original function.

2.2.5 RQ5: How can modifications to an impeding function be validated for soundness and improvement to fuzzing?

Instead of validating patches for soundness before application, patches are applied eagerly and validity testing is done in two stages after variant binaries have been generated. First, each input in the fuzzing input queue that was used to generate the variant binaries via the previous stages is run on each variant binary. Any variant that experiences a significant unsound behavior such as crashing or hanging is immediately discarded. Variants that successfully run with the original set of inputs are placed under fuzzing for a brief period (parameterized in REFACE by the length of the period). Crashes during this period do not necessarily represent unsoundness, as they could potentially reflect newly discovered logic after the binary's execution proceeds through the original stuck point. However, a crash could also be due to an invalid patch, so after the brief period of time (generally, on the order of minutes) inputs from the new fuzzer job's input queue are run on the binary using the coverage tool `llvm-cov`. The callstacks of any crashing locations are examined and any crashes inside of patched code are taken to indicate an unsound patch. The remaining variant binaries are kept and fuzzed for a longer period of time to generate a larger set of inputs to increase the chance of generating an input that can be re-hosted to allow the original binary to pass through the stuck point.

3. EVALUATION

3.1 Best Practices

Evaluation of fuzzers specifically is a highly active research area with little agreement among researchers on a precise way to measure fuzzer performance or compare fuzzers against each other. This problem is compounded when considering a project such as REFACE, where the majority of the process remains identical between a typical configuration and the implemented tool with a single variable modified. In this case, the variable modified is the binary itself, which gives rise to additional complexity in evaluating performance, especially relative performance to other approaches. Despite the lack of formal settled specification for performing evaluation, some consensus has been established. Generally, evaluation methods for fuzzing differentiate themselves in a select few categories. The first, and perhaps most important, is the primary and secondary metrics used to evaluate effectiveness. Of particular note for this work is the lack of recommendation of any paper surveyed for fuzzer configuration when evaluating properties of a fuzzer other than maximum bugs found, or for evaluating fuzzers in a manner other than a "head to head" configuration to drag race fuzzers for supremacy. FuzzBench [103] aims to directly provide these configurations as a benchmarking service, and also provide reproducibility, multiple runs, deduplication, and a selection of programs as benchmark targets. FuzzBench does, however, suffer from overfitting, as the benchmark components are known and can be tested and improved against. We outline several recommendations from literature for fuzzer evaluation and discuss whether we will utilize the recommendations as is, or give a reason for adopting a different approach.

3.1.1 Comparison Metrics

As discussed in Subsection 1.1.4, many metrics exist for the continuous evaluation fuzzers use to mutate input seeds. Often, these include some combination of edge, branch, and path coverage. These three metrics are subtly different. Given the program control flow graph in Figure 3.1, edges are easy to identify. Any edge in the graph is also an "edge" by the

common definition used to describe program coverage. That is, the edge from block 0 to block 1 is an edge, as is the edge from block 3 to 6, and the edge from block 5 to 4. *Branch* coverage, in contrast, is only concerned with nodes with multiple outdegrees. In this case, there are two branches from blocks 2 and 4. 100% branch coverage in this graph means both edges from both blocks 2 and 4 are covered, a total of four edges. Thus, branch coverage represents a subset of the possible edge coverage in the program. *Path* coverage, in contrast to edge and branch coverage, is context sensitive. The path context depends on the implementation, but assume for example the size of the context here is 7 blocks. The path from $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ would be one path with a context size of 5, while the path from $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 5$ represents another with a context size of 7. This second example reveals a fundamental limitation of path-based context tracking. Paths that traverse the same code multiple times greatly increase both the size and number of paths needed to represent all paths in a given segment of code. More practically, many fuzzers do not support path-based coverage tracking. Branch and edge coverage are both useful, however, and are used in many fuzzers including AFL [77].

As a metric for comparison, coverage has proven contentious, with several prior works arguing for and against coverage as a metric for evaluating fuzzers. In 2015, Kochhar et al. [104] presented a study of code coverage as an indicator for effectiveness in large programs, which in this case refers to Apache HTTPClient and Mozilla Rhino, both of which are real-world programs with a large amount (hundreds of thousands of lines) of code. The authors discovered a moderate correlation between coverage and bug discovery in these programs. Interestingly, the authors observe this correlation for "both statement and branch coverage". Also of interest is the claim cited by Kochhar, but put forth by differently by Just [105] et al. Kochhar claims that "it is not clear whether the effectiveness of a test suite in killing mutants is representative to its effectiveness in killing real bugs". However, in the cited work Just claims that "results show a statistically significant correlation between mutant detection and real fault detection". Just also presents some limitations, however, including that many real bugs did not have corresponding classes of bugs represented by the injected faults, or mutants, they inserted into the tested code. Klees et al. [106] presented one of the

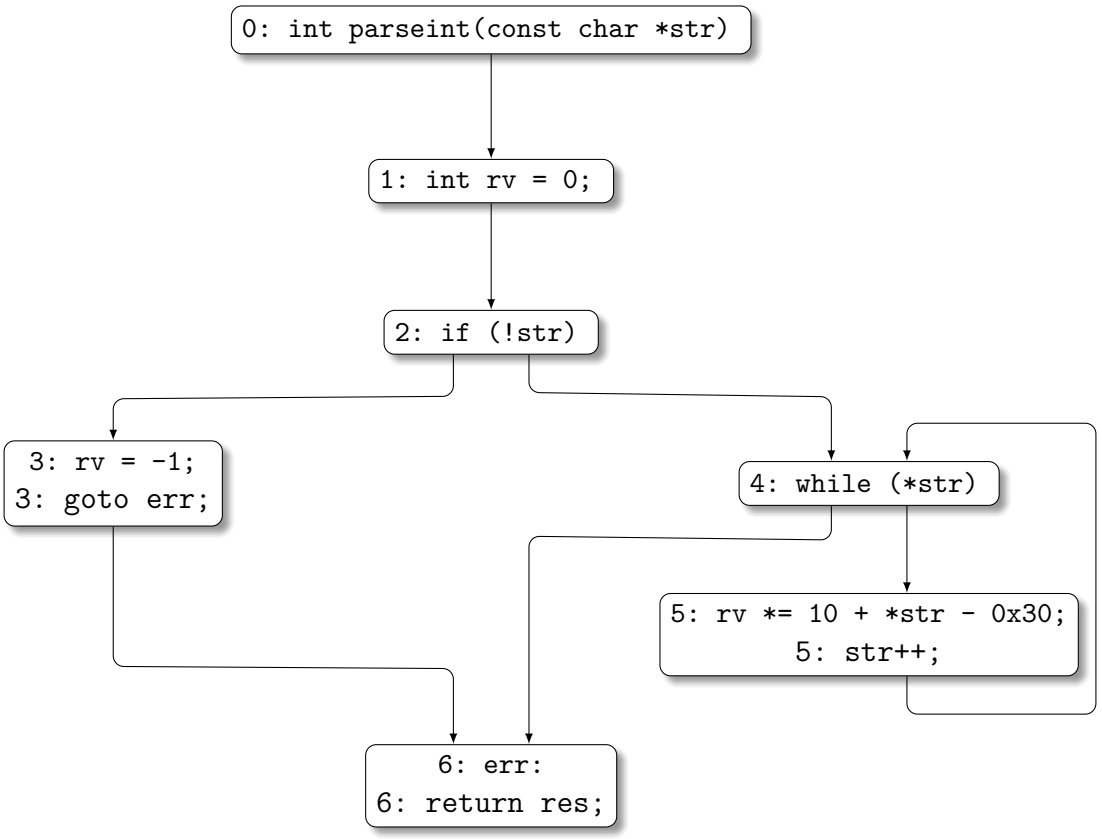


Figure 3.1. An example control flow graph to illustrate various coverage metrics.

most important works in fuzzer evaluation, *Evaluating Fuzz Testing*, a survey work intended to make recommendations for fuzzer evaluations based on concrete data. Shockingly, their work found problems by their criteria with 32 separate fuzzing papers published prior. They note that "14 out of 32 papers we examined used code coverage to assess fuzzing effectiveness", but reference Inozemtseva [107]'s work which found a very weak correlation between coverage and fuzzer effectiveness. However, Inozemtseva found this correlation on large Java programs, where code and data interact differently than they do in C programs. Furthermore, there is a stark contrast in the high correlation between coverage and effectiveness when test suite size – or the size of the software under test – was ignored, and the much lower correlation when size is considered. Inozemtseva also uncovered a significant conclusion that statement, branch, and path coverage show little difference between them. Klees tested all 32 aforementioned fuzzing projects with the same configuration to identify weaknesses in the various papers' approaches. They suggest that bugs found should always be used as the primary metric when evaluating fuzzers, because ultimately a fuzzer is only useful if it successfully uncovers bugs. They suggest that when evaluating fuzzers with a "bugs found" metric, the ground truth should be used if at all possible. In addition, they note that there is weak or no correlation between the number of crashing input seeds found and the number of unique crashes, and assert that bugs must be at a minimum deduplicated when presented as data points. With respect to code coverage, Klees maintains that it is useful as a secondary metric, but asserts bug finding should be prioritized. A newer work from Bohme [108] presents new evidence directly contrary to that of both Klees and Inozemtseva. The work shows conclusively that coverage is strongly correlated with bug finding, however the authors make careful note that it should not be used to compare fuzzers to each other to determine "superiority". Instead, they suggest using an agreement test to determine whether a fuzzer finds more bugs *and* achieves more coverage.

3.1.2 Seed Selection

Fuzzers rely on input seeds for all exploration of target binaries. Klees et al. find a discrepancy between testing with an empty (or null) seed as opposed to a provided "good" seed specifically tailored to the target program. They suggest where possible testing should be done using *both* a "good" and "bad" seed. Seeds tailored to a program typically contain data that is already formatted to conform to the expected input format of the binary. For example, a black-box program that decodes an unknown data format may be very difficult or time consuming to construct seeds for. In addition, fuzzing harnesses for both open source and black box programs often harness specific portions of code, not the entire program. An image parsing library may have fuzzing harnesses included, but these harnesses may exercise only the re-encoding portions of the library. Because of this, even code that is already set up with fuzzing supported may not provide "good" seeds for security evaluation. In addition, external analysts evaluating binary programs for security flaws generally must either create seeds themselves through manual effort, which requires time and money, or begin fuzzing using "bad" seeds in the hopes of passing through enough checks via the mutation or generation processes. Therefore, there are compelling reasons aside from performance differences to evaluate fuzzing projects over both "good" and "bad" seeds. Performance is the key consideration, however, and the same fuzzer may perform very differently when run using different seeds.

3.1.3 Evaluation Time and Repetitions

Most importantly, Klees et al. recommend using multiple runs of any evaluation to reduce the effect of non-determinism or "good" and "bad" runs. They do not give a specific number as a recommended minimum, instead recommending runs be performed until a statistical test shows a significant result. They also suggest that any fuzzing run be configured to run for at least 24 hours. The primary motivation of fuzzer evaluation is to compare mutation or generation algorithms, which they note may not begin to produce desirable results until after a certain time has elapsed. In addition, they note that evaluation over shorter timelines can

be simulated by simply excluding data after a certain time in the fuzzing cycle. In general, fuzzing time is a problem for repeatability of results, especially if the time is not given or evaluated.

3.1.4 Fuzzer Configuration

Klees et al. place little emphasis on fuzzer configuration in their work. In order to accurately assess algorithms, they state (and we agree), that fuzzers must have an accurate head-to-head configuration. For example, there is little value in a benchmark of one fuzzer running on 8 cores compared to another fuzzer running on a single core, unless the discrepancy is the point of the benchmark. This is a difficult property to express empirically, however, and Klees et al. give no concrete numbers or recommendations. For example, it is obvious that giving one fuzzer additional CPU resources over another gives that fuzzer an unfair advantage, but some fuzzers such as AFL++ posses numerous options and toggles. AFL++ specifically has options to enable features from previously discussed projects that make massive leaps in performance, such as REDQUEEN. Klees et al. do not present specific recommendations, but they do mention that evaluation should be as fair as possible and fuzzers should not be intentionally reduced in effectiveness when configured for evaluation.

3.1.5 Datasets

Datasets suitable for fuzzing research are difficult to create, as they must satisfy several requirements. First, they must be publicly available, and fuzzing must be able to be reasonably performed against them. Often in the real world, fuzzing requires creation of a harness, which is a time consuming process. Next, they must contain bugs and the bugs must be known ahead of time in order to evaluate, as discussed above, against the ground truth. Very few purpose-built test sets exist, of which the dataset from the Cyber Grand Challenge (CGC) [25] is the largest. It consists of over 200 small to medium programs with widely varied behavior and implementations. All known bugs in the CGC dataset are doc-

umented, and therefore can be very efficiently triaged. However, the CGC dataset consists almost entirely of programs under 10,000 lines of code, and is therefore not representative of "real-world" fuzzing applications. In addition, although CGC consists of over 200 binaries, Klees et al. note that Driller, VUzzer, and Steelix evaluate on under half of the binaries. Injection of bugs into existing programs is therefore a popular tactic: large real-world software can be used as a target to insert known errors into, in order to take advantage of ground truth knowledge while also presenting the fuzzer with a large binary. Dolan-Gavitt et al. also investigated realistic bug injection methodologies. The result is the LAVA-M [109] benchmark, which addresses some of the identified problems with fault injection identified by Just. This benchmark scales bug injection and allows extraction of the ground truth from the set of injected bugs. However, it does not satisfactorily address the problem of realism and injects only trivial and semi-trivial buffer overflow vulnerabilities where a value must be set to expose the vulnerable code. Magma, presented by Hazimeh at al. [110], aims to solve this limitation by utilizing real bugs from the past and "re-inserting" them into real programs that can be re-analyzed to rediscover the previously extant bugs. This allows for reproducibility and realism, along with verifiability to allow less costly triaging of results. Magma also has an advantage in its human curation, as opposed to LAVA-M and other injection works that use automation to inject bugs but, according to Dolan-Gavitt, the bug injection itself can cause instability or further bugs and it can be difficult to ascertain the root cause of a bug discovery in an injected program. FIXREVERTER [111], a more recent work, modified the bug injection paradigm. Instead of finding code locations to insert bugs, FIXREVERTER identifies locations where a bug would exist if a condition was met, and creates an opportunity for that program to exist. They check "bug fix patterns" using static analysis, then modify the binary to enable the bug. This method implements few bug types, but provides a very convincing argument toward avoiding injection of brand new bugs, instead of reusing old bugs. Most significantly, this creates a "realism" factor.

3.2 Testing Methodology

We attempt to include as many recommendations by the authors listed above as possible. The primary difference between this work and other projects in fuzzing research that are included in the aforementioned fuzzer benchmarks, such as AFL++ or Honggfuzz [112], is scope. REFACE does not attempt to be a fuzzer in itself, rather it attempts to augment an existing fuzzer and enable it to surmount difficult code paths to make faster progress deeper into the binary. Instead of comparing against multiple other fuzzers, as a typical fuzzing project might, we only consider a single fuzzer: AFL++, and compare AFL++ using our approach against AFL++ with no extraneous assistance. In addition to the decision to use AFL++, several other choices were made to strike a balance between the unique testing requirements of REFACE and the recommendations and best practices outlined above.

3.2.1 Comparison Metrics

The hypothesis of this work is essentially that REFACE will (a) enhance coverage, and the *speed* of achieving enhanced coverage, on appropriate binary programs during fuzzing, and (b) that this gain in coverage will enable a fuzzer to discover bugs more quickly. To this end, the primary evaluation mechanism we use is *edge* coverage. *Edge* coverage is roughly equivalent to branch coverage, and is the metric used by AFL++ to calculate whether inputs have activated additional coverage. We also analyze any discovered bugs, as well as the time required to discover them. In general, "time to first bug" is an imperfect metric, however as a key goal of this work is to enhance not only the coverage and bug finding ability of fuzzers but to enhance the speed with which they analyze black-box binaries with little preparation time, we feel this time-based metric is appropriate.

3.2.2 Dataset

This is a prototype implementation of REFACE, and is capable of analyzing light to medium weight programs. However, REFACE is not a general-purpose tool in terms of applicability. As discussed in Subsection 2.2.1, it depends on the presence of impeding functions in programs to achieve greater performance. Therefore, to achieve the greatest possible diversity in the dataset, we focus efforts on the CGC dataset of challenge binaries. Several stages of analysis led to this decision, first among them a static analysis pass using the CodeQL [113] code scanner to evaluate whether the CGC met the needs of evaluating the effectiveness of REFACE. CodeQL facilitates writing "queries" that run over a project's code, searching for locations where particular patterns appear. It also incorporates powerful static control and dataflow analysis capabilities. We implemented queries for each of the criteria listed above to discover a ground truth over the dataset of functions that *may* verify the search criteria.

First, we filtered for functions satisfying basic prerequisites such as non-library and non-builtin definitions. Next, functions were checked for side-effect freeness. In the context of the Cyber Grand Challenge binaries, a function was defined side-effect free if it did not call any of the five defined system calls for the dataset and did not make any writes to *global* data. This is a restrictive assertion that was lightened (as discussed previously) in the binary-only assertion system. However, because source code is available for CodeQL queries, we opted for higher accuracy rather than lower false negative rates. Next, functions were checked to ensure they could feasibly return multiple different values. To do this, local dataflow was examined for each function. CodeQL's speed allowed for examining local dataflow sources defined as *each* other expression in the function, while the sinks were defined as each return statement in the function. Then, the set of possible expressions whose values flow into the return statements was checked to ensure at least two possibilities existed for a given function. Next, another local dataflow analysis with sources defined for each return statement and sinks defined on each comparison expression allowed testing the criteria that control flow is modified by the return value of the impeding function. Finally, *global* dataflow allowed

checking whether a function received part of an external input as an argument. This dataflow analysis, more than the local analyses checked prior, is imprecise. CodeQL implements a *may* analysis for dataflow, meaning unlike with `libdft`, implicit dataflows are captured. This affords more power to the analysis, but also provides room for error because REFACE does not handle implicit dataflow. The results of the CodeQL analysis were utilized to select targets appropriate for testing with REFACE as well as to verify the output of REFACE and validate the effectiveness of testing the same assertions with only a binary program instead of full source code.

From the CGC dataset, we selected four binaries best suited to analysis using REFACE, using functions that perform many checks on user data, copy user data, or transform it. These programs all contain at least one function identified by both CodeQL and manual analysis as an opportune impeding function. The selected binaries were:

- CGC_Symbol_Viewer_CSV
- Mathematical_Solver
- Loud_Square_Instant_Messaging_Protocol_LSIMP
- Parking_Permit_Management_System_PPMS

These data points were selected without regard to the performance of AFL++ against them, as it is a general purpose fuzzer and therefore able to handle nearly any program as a test subject. A larger dataset would be ideal, however these four binaries serve as exhibits of the effectiveness of the concept presented here. A larger evaluation is planned, but given to future work as a function of manual analysis resources required to interpret results of large-scale testing.

3.2.3 Seed Selection

Seed selection is a key consideration, and we pay special attention to seed selection due to the specific nature of this approach. In general, we hypothesize REFACE's approach is most

helpful when well-formatted seeds are *not* available, as manually crafted seeds may already bypass many impeding functions that would otherwise impede progress of the fuzzer. However, in order to accurately assess the applicability of the methods described, it is useful to observe the *discrepancy*. To avoid bias toward either approach, we test each binary program with both a set of 20 crafted seeds provided in the CGC dataset along with the programs. We also test each binary with a *null* seed composed of a sequence of 32 zeroes. This allows a comparison of the same approach in both scenarios.

3.2.4 Evaluation Time and Repetitions

Fuzzing time is a second critical procedural choice. Klees et al. clearly recommend a fuzzing run of a least 24 hours, and we adopt this time length as well. For the purposes of this evaluation, the majority of exploration will, we hypothesize, occur early in the fuzzing process, with impeding functions impeding further exploration after a short time. However, to preclude the possibility of the fuzzer discovering new "interesting" inputs significantly later in fuzzing, we implement the full time. More realistic time recommendations for evaluating systems that do not implement fuzzing components but rather implement an assistance mechanism for a fuzzer are discussed in Chapter 4. In addition to fuzzing for 24 hours, we make multiple repetitions of each test in order to preclude testing outliers or random differences in the progression of the mutation engine. Each binary is run in three full separate fuzzing cycles, and each separate cycle is also tested with separate invocations of REFACE during the runtime of the fuzzer, creating three fully separate experiment repetitions. These repetitions address the threat to validity that a given observed run is a spurious maximum or minimum that could make it difficult to accurately compare results.

3.2.5 Fuzzer Configuration

Finally, for scalability of testing as well as reproducibility of results, each fuzzing "job" was run on one CPU core, using the default settings for the AFL++ fuzzer. The fuzzer

was run in binary-only fuzzing mode, which executes using the QEMU emulation system. Using binary-only fuzzing mode is reflective of real workloads we hypothesisze REFACE is best suited for, where there is little visibility into or knowledge of a target binary. These fuzzing jobs were carried out in parallel across the cores of a single server, and at a configured interval (in this case, 30 minutes), the REFACE pipeline was invoked with the *current* fuzzer queue as input. The pipeline, executed at each interval, identified impeding functions matching the criteria and produced variant binaries suitable for fuzzing. If any new variant binaries were produced by a given invocation, a new fuzzing process was created for each, with an initial input queue of the current input queue of the primary fuzzer. On exit of a variant binary fuzzing job, the current fuzzer queue was captured for analysis. Finally, after 24 hours, all fuzzing jobs were stopped and outputs were captured.

3.3 Test Results

3.3.1 Coverage

Coverage results show a strong support of the first half of the hypothesis regarding bugs or coverage found, it gives insight to examine auxiliary metric results first. As mentioned above, the first data collected to analyze the performance of the approach was the fuzzing queue from AFL++, complete with metadata including timestamps. The queue was collected for each primary fuzzer, each of which ran for 24 hours, as well as from variant binary fuzzing jobs, each of which ran for only 30 minutes. Figure 3.2 displays the number of total inputs in the queue as a function of time. For both the "good" and "bad" starting seed configurations, we observe an exceedingly sharp drop-off of new input discovery after a relatively short amount of time. In general, sharp drops in new input discovery can indicate a lack of forward progress made by the fuzzer, a problem that indicates the binary may be becoming stuck at a check it cannot bypass by stochastic approaches. Figure 3.2 shows that this input discovery effectively drops to zero by approximately 300 minutes of fuzzing although by far the largest decrease occurs within the first 30 minutes. Interestingly, this drop-off occurs

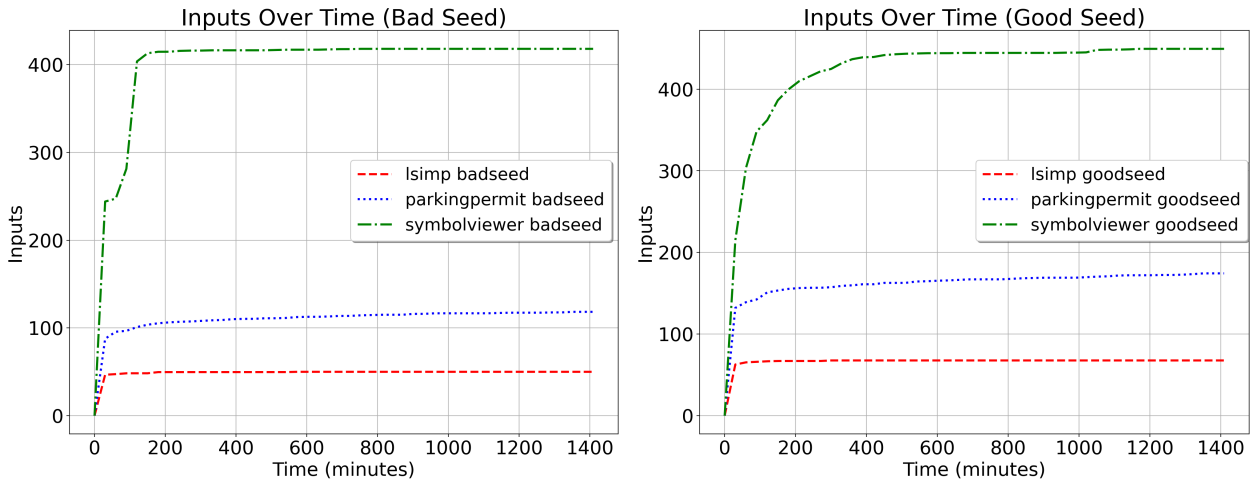


Figure 3.2. Newly discovered inputs added to the AFL++ queue over time.

similarly for "good" and "bad" seeds, and occurs over all repetitions.

Coverage is the key metric we aim to *increase* using REFACE, so analysis of coverage during the run of the fuzzer as shown in Figures 3.3 and 3.4 proves highly interesting. Unlike the graph of new inputs discovered over time, which shows only a single input, if any, discovered at a time for the majority of the fuzzing run, coverage discovery remains significantly higher through the fuzzing process, with less of a drop-off for both the `Parking Permit` and `CGC_Symbol_Viewer_CSV` binaries. First, consider the "good seed". Small (single input) numbers of additional discoveries barely visible in Figure 3.2 correspond to large jumps in new coverage discovery. This seems to imply that late coverage, likely coverage reaching *deeper* parts of the program, is as valuable to expanding coverage as early coverage discovered at the beginning of a test. Perhaps even more so, as single input discoveries later in the runtime of the fuzzer account for basic block discovery on par with that triggered by *multiple* inputs provided as initial seeds. These two observations imply that REFACE should indeed be effective at enhancing the coverage of the fuzzer. In fact, considering the graphs of coverage as compared to the graph of newly discovered stuck points, we observe that stuck point discovery is nearly identically correlated with coverage discovery, implying that nearly all branches discovered are in fact stuck points. This gives weight to the *additional* criteria for selecting impeding functions, as clearly selecting every stuck point would result

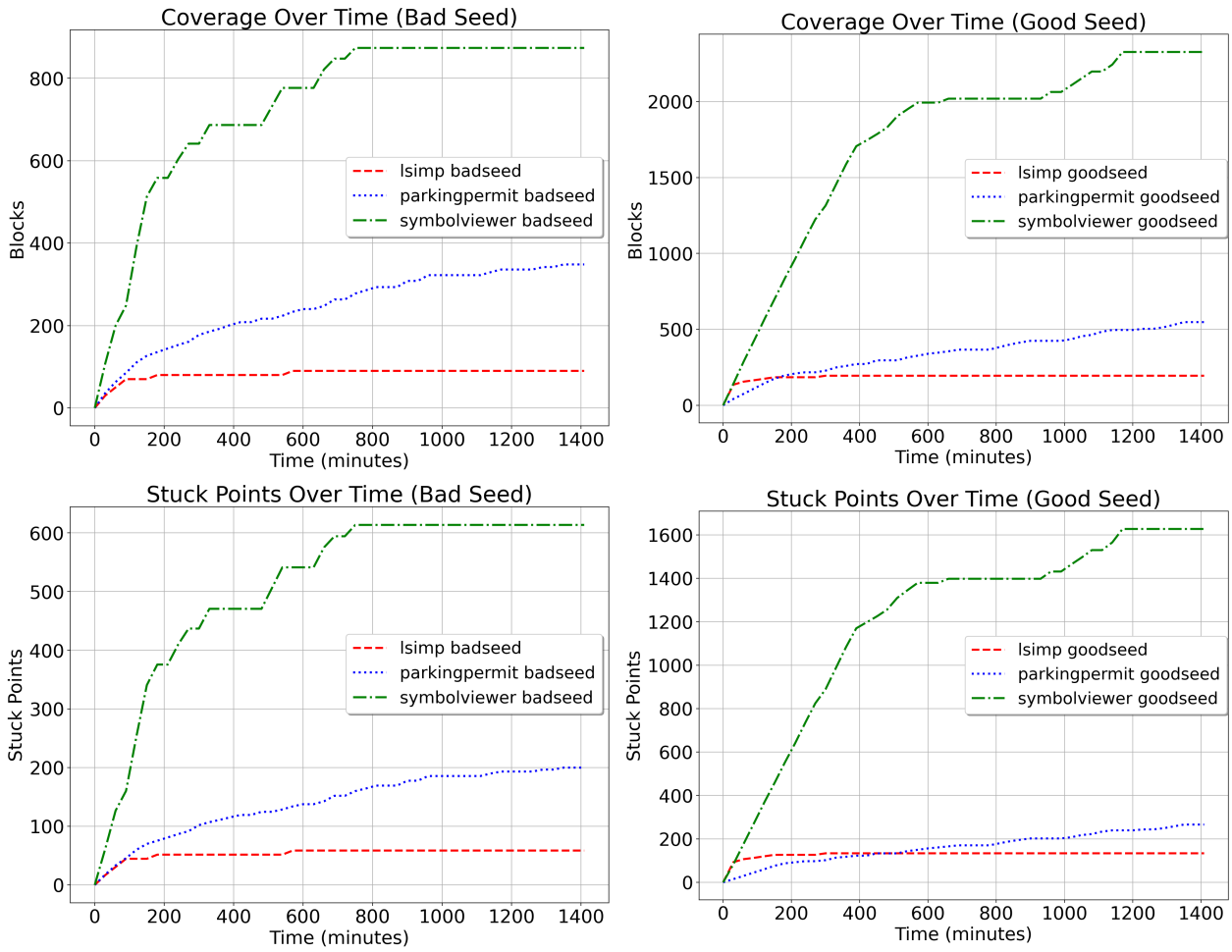


Figure 3.3. Newly discovered coverage and stuck points.

in numerous false positive results.

Recall that a "candidate function" or impeding function is a function verifying *all* of the criteria identified previously. That is, an impeding function receives user input, and causes a stuck point to occur via an imbalanced return value distribution. The identification of an impeding function in the binary is correlated with the generation of a variant binary and the subsequent fuzzing of that binary. Consider Figure 3.5. It is clear the vast majority of candidates are identified relatively quickly, with between 1 and 5 variant binaries created for each original binary within the first interval of running REFACE. In the *Parking Permit* binary, further coverage later in the runtime facilitates discovery of additional impeding functions

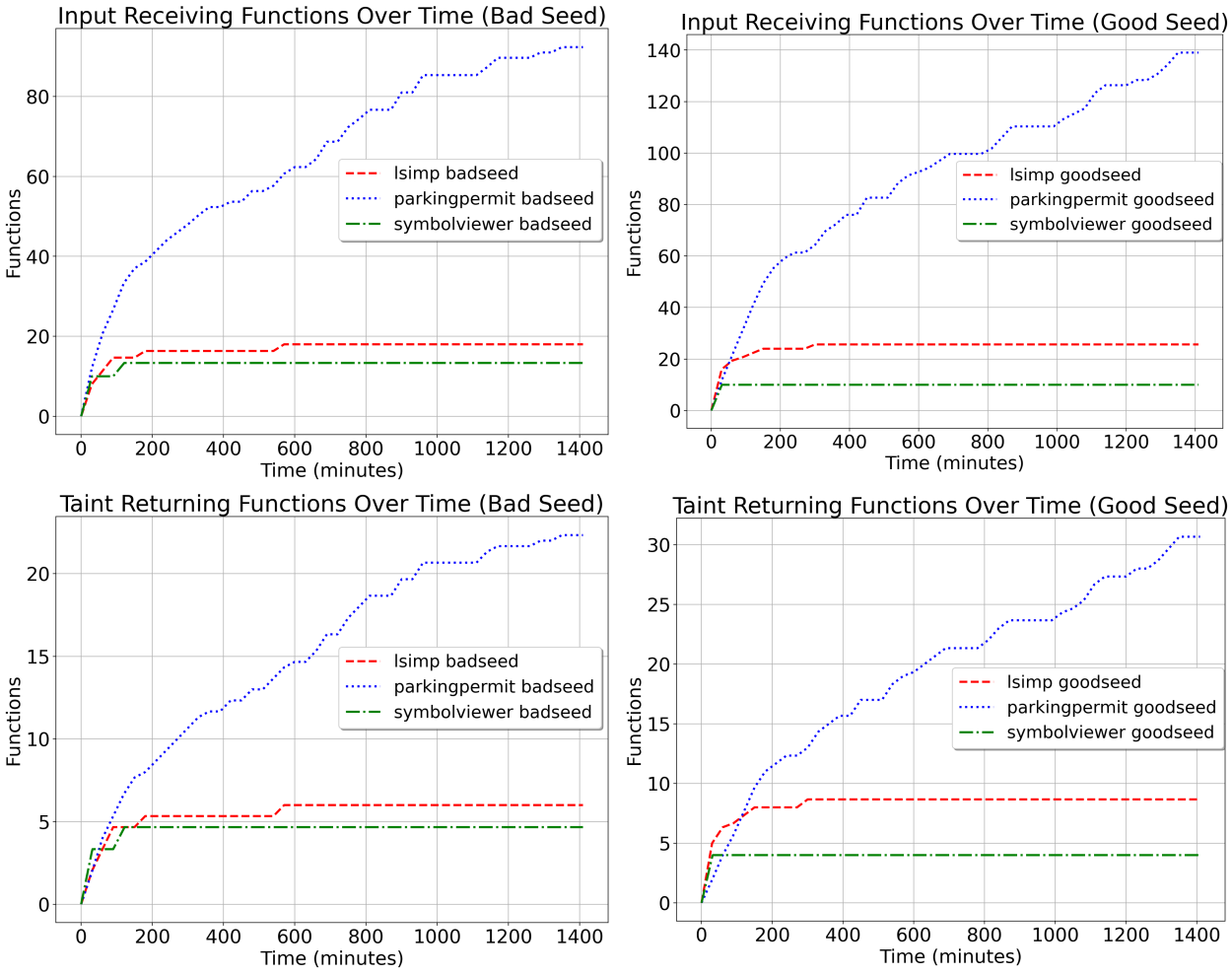


Figure 3.4. Newly discovered input receiving and stuck point creating functions over time.

and variant binaries, but this is not the case for the other binaries. However, the other binaries all produce a larger number of *initial* variant binaries with the notable exception of the `PKK Steganography` binary when executed with a "bad" seed. The correlation between increases in coverage and discovery of new impeding functions is encouraging. In an ideal scenario to prove the usefulness of REFACE, the fuzzer would *always* become stuck due to an impeding function, and not due to a non-impeding function-caused stuck point.

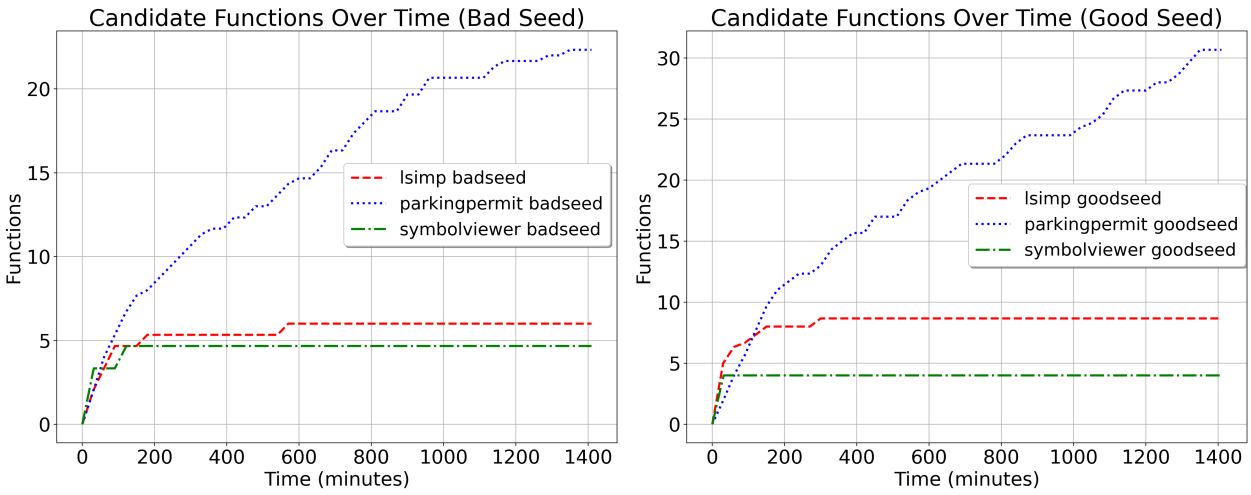


Figure 3.5. Newly discovered candidate functions over time.

The above conclusions suggest that REFACE is effectively locating impeding functions during runtime, and also suggests the discovered impeding functions are indeed a significant blocker to advancement by the fuzzer. This is a promising result for the validity of the first component of the approach given here for identifying functions as abstract locations blocking progress when fuzzing a binary. In addition, the REFACE system runs alongside an existing fuzzer to obtain these results and uses little compute power relative to the fuzzer itself, making this approach relatively "cheap" for locating functions as areas blocking progress. However, the *key* result is not necessarily whether impeding functions can be identified and successfully modified, but whether doing so materially enhances the fuzzing process. In fact, we demonstrate a positive result, with some significant caveats and prior work. However, in general we show this method does increase coverage of the target binary during fuzzing. In addition, this method obtains increased coverage of the target binary *well* before the regions

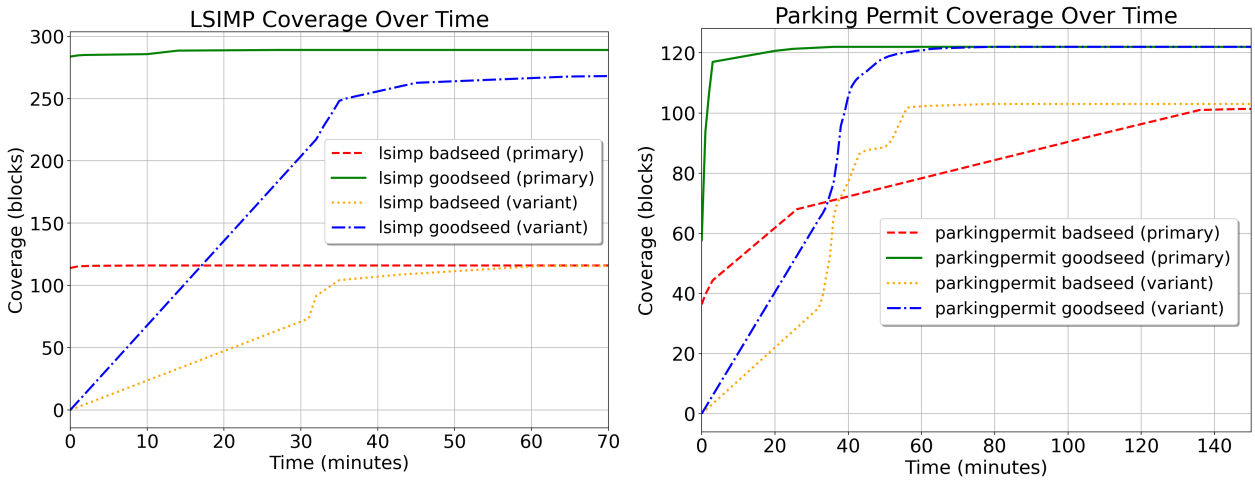
Table 3.1. Table of maximum coverage gained by patched program through a single runtime.

Target	Seed Type	Run #	Time (min)	Coverage Gain (blocks)
parkingpermit	badseed	1	57	6
parkingpermit	badseed	2	56	9
symbolviewer	badseed	1	53	303
symbolviewer	goodseed	1	92	257
symbolviewer	goodseed	2	59	142
symbolviewer	goodseed	3	41	273

of code under test can be explored by an unmodified fuzzer.

Consider Table 3.1 above. The table lists each binary under test with a successfully executing patch (`PKK_Steganography` experienced failures to analyze the type of impeding function, a shortcoming discussed in the future work section), along with the category of seed and the trial number. The final column displays the "maximum patch coverage", or the comparison at any given point in time of the number of basic blocks covered by the patched binary compared to the unpatched binary. The method for calculating this metric was to iterate over each time step for both patched and unpatched binaries and check the size of the set difference of coverage between them. Then, the largest difference is reported here. This is a best-case metric, but demonstrates the exact hypothesis we sought to demonstrate by obtaining faster time to reach a larger coverage than the unpatched binary. This coverage leap at an earlier time than the original binary can be seen in graphs of variant versus original binary performance in Figure 3.6.

Figure 3.6 displays coverage graphs of original versus variant binaries over time. While coverage over time is similar, we note that two out of three cases, REFACE achieves faster time to reach a plateau of coverage improvement, and in all cases the fuzzer achieves nearly identical maximum coverage in the variant binary. In all cases the replacement function contains less basic blocks than the patched function, so this coverage is not due to the applied patch. This increase in both speed and coverage amount is modest, but displays a confirmation of the section of the hypothesis regarding coverage increases as a result of modifying



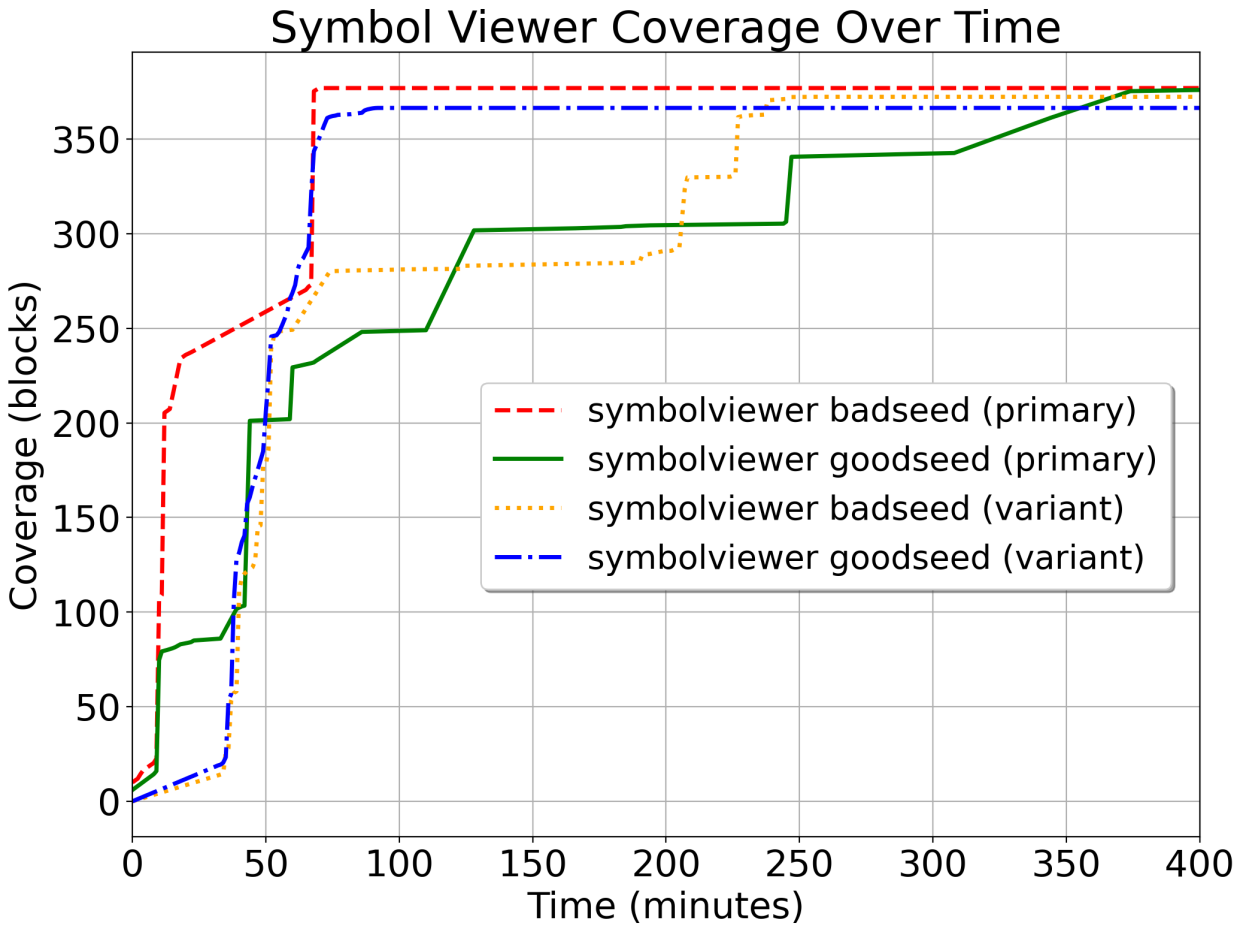


Figure 3.6. Coverage of original binaries compared to variant binaries over time.

impeding functions. In particular, variant binaries of the `GCG_Symbol_Viewer_CSV` binary reach greater coverage than the good-seed fuzzer on the original binary, a case explored in greater depth in Subsection 3.3.4. Variant fuzzing also reaches a coverage plateau more quickly on the `Parking Permit` binary with a bad seed.

3.3.2 Impeding function Identification

As discussed in Subsection 3.2.2, we use CodeQL to statically identify possible impeding functions with static source analysis. In order to empirically measure how effective REFACE is in identifying impeding functions, we compare the actually detected functions with applied patches to the "ground truth" set of functions from CodeQL. In almost all cases, CodeQL provides a significant over-estimate of the number of real impeding functions present in a binary. There are several reasons for this. First, and most impactful, is the lack of dynamic information available to CodeQL. This means it is unable to determine when a function returns one value more often than another, and is therefore unable to verify **C3**, the assertion that the distribution of return values must be skewed. In addition, although CodeQL implements dataflow analysis, it is an estimation, and tends to raise many false positives. The following tables Table 3.2, 3.3, 3.4, and 3.5 display the set of functions identified by CodeQL as compared to the set of functions identified during the runtime of REFACE and patched. Note that each function may be patched more than once, as different runs may expose different identifications of buffer locations and sizes.

In all cases, REFACE was able to identify at least one common impeding function with CodeQL. In `LSIMP`, `PKK_Steganography`, and `CGC_Symbol_Viewer_CSV`, REFACE was able to identify a manually verified real impeding function CodeQL was unable to identify. Via manual verification, we conclude the majority of functions recognized by CodeQL are false positives for this analysis. Primarily, this is due to a lack of biased return value distribution. For example, all `is` type functions such as `isascii` do not exhibit this characteristic over well-distributed input from the fuzzer. A small number of functions recognized by CodeQL

Table 3.2. Impeding functions reported by CodeQL compared to REFACE for PPMS

Function	CodeQL	REFACE
cgc_allocate_new_blk	✓	
cgc_calloc	✓	
cgc_fdprintf	✓	
cgc_isalnum	✓	
cgc_isalpha	✓	
cgc_isascii	✓	
cgc_isdigit	✓	
cgc_islower	✓	
cgc_ispunct	✓	
cgc_isspace	✓	
cgc_isupper	✓	
cgc_isxdigit	✓	
cgc_malloc	✓	
cgc_memchr	✓	
cgc_memcmp	✓	✓
cgc_memcpy	✓	
cgc_permit_new	✓	
cgc_permit_test	✓	
cgc_pring_test	✓	
cgc_read_n	✓	✓
cgc_read_until	✓	
cgc_realloc	✓	
cgc_strcasecmp	✓	
cgc_strchr	✓	
cgc_strcmp	✓	
cgc_strdup	✓	
cgc_strlen	✓	
cgc_strncasecmp	✓	
cgc_strncmp	✓	
cgc_strsep	✓	
cgc_strtol	✓	
cgc_strtoul	✓	
cgc_tolower	✓	
cgc___validate_license_number	✓	
cgc___validate_permit_token	✓	

Table 3.3. Impeding functions reported by CodeQL compared to REFACE for LSIMP

Function	CodeQL	REFACE
cgc_calloc	✓	
cgc_compute_guard	✓	✓
cgc_decode_data	✓	
cgc_fdprintf	✓	
cgc_isalnum	✓	
cgc_isalpha	✓	
cgc_isascii	✓	
cgc_isdigit	✓	
cgc_islower	✓	
cgc_ispunct	✓	
cgc_isspace	✓	
cgc_isupper	✓	
cgc_isxdigit	✓	
cgc_malloc	✓	✓
cgc_memchr	✓	
cgc_memcmp	✓	
cgc_memcpy	✓	
cgc_parse_msg		✓
cgc_process	✓	
cgc_read_n	✓	✓
cgc_read_until	✓	
cgc_realloc	✓	
cgc_strcasecmp	✓	
cgc_strchr	✓	
cgc_strcmp	✓	
cgc_strdup	✓	
cgc_strlen	✓	✓
cgc_strncasecmp	✓	
cgc_strncmp	✓	
cgc_strsep	✓	
cgc_strtol	✓	
cgc_strtoul	✓	
cgc_tolower	✓	

Table 3.4. Impeding functions reported by CodeQL compared to REFACE for PKK Steganography

cgc_allocate_new_blk	✓	
cgc_calloc	✓	
cgc_embed_text	✓	
cgc_extract_text	✓	
cgc_isalnum	✓	
cgc_isalpha	✓	
cgc_isascii	✓	
cgc_islower	✓	
cgc_ispunct	✓	
cgc_isspace	✓	
cgc_isupper	✓	
cgc_malloc	✓	✓
cgc_memchr	✓	
cgc_memcmp	✓	
cgc_memcpy	✓	
cgc_output_pkk	✓	
cgc_parse_input	✓	
cgc_read_n	✓	✓
cgc_readuntil	✓	
cgc_realloc	✓	
cgc_skip_whitespace		✓
cgc_strchr	✓	
cgc_strcmp	✓	
cgc_strlen	✓	
cgc_strsep	✓	
cgc_strtol	✓	
cgc_strtoul	✓	
cgc_tolower	✓	

Table 3.5. Impeding functions reported by CodeQL compared to REFACE for CGC Symbol Viewer CSV

cgc_allocate_new_blk	✓	
cgc_calloc	✓	
cgc_cgcf_is_valid		✓
cgc_cgcf_parse_file_header	✓	
cgc_cgcf_parse_section_header	✓	✓
cgc_fdprintf	✓	
cgc_isalnum	✓	
cgc_isalpha	✓	
cgc_isascii	✓	
cgc_isdigit	✓	
cgc_islower	✓	
cgc_ispunct	✓	
cgc_isspace	✓	
cgc_isupper	✓	
cgc_isxdigit	✓	
cgc_malloc	✓	✓
cgc_memchr	✓	
cgc_memcmp	✓	
cgc_memcpy	✓	
cgc_read_n	✓	
cgc_read_until	✓	
cgc_realloc	✓	
cgc_sl_insert	✓	
cgc_strcasecmp	✓	
cgc_strchr	✓	
cgc_strcmp	✓	
cgc_strdup	✓	
cgc_strlen	✓	
cgc_strncasecmp	✓	
cgc_strncmp	✓	
cgc_strsep	✓	
cgc_strtol	✓	
cgc_strtoul	✓	
cgc_tolower	✓	

would eventually become true positive results, but in the tested binaries over the 24 hours tested, they were not reached in a context in which they would become true impeding functions. Finally, many functions are statically-compiled library functions and are unused in the binaries. For example, `LSIMP` contains no calls to `cgc_strcasecmp`. Some functions, such as `cgc_read_n` would appear false positives upon first consideration, however `cgc_read_n` is in fact a true positive result in the context of the chosen binaries. In `LSIMP`, `cgc_read_n` is called in a loop, and data is read into the same buffer each iteration. Its return value is checked, and the loop is exited if no data is read, which only occurs once out of all read operations. Thus, it creates a stuck point and receives user input if called more than a single time on the same buffer. In addition, a patch to this function can exercise additional control flow by potentially reading more than requested into the buffer, introducing new potentially behavior. Via manual analysis of identifiied impeding functions, we find REFACE discovered the first encountered impeding function with 75% accuracy over the programs tested. REFACE did not identify `cgc_parse_input` in `PKK_Steganography`, as it does not *receive* user input through a parameter. Instead, it performs read operations from standard input in its function body, then performs parsing immediately. Despite not conforming to the current definition of a impeding function, we recognize this as a valid case and leave implementation of additional checks and criteria to detect similar cases to future work.

3.3.3 Crashes

Using default AFL++ without Reface, the fuzzer experiences various levels of success. Over the four binaries evaluated, `LSIMP` and `Parking Permit` experienced zero crashes each over the *entire* 24 hour fuzzing run. Several crashes were reported for `LSIMP` by AFL++, but these were due to AFL++'s qualifications for a reported crash, which includes any instance where a binary exits with a signal. `LSIMP` exits with a signal during normal operation, which causes AFL++ to report these exits as crashes, but a triaging session using GDB shows these are false positives. PKK Steganography contains an unintended SIGFPE crash (the crash is not listed as ground truth in the CGC documentation) when dividing by zero in

its pixel parser, which divides a constant by both the width and height, both of which are user-provided. This crash is discovered almost immediately (after 34 seconds), and is the only crash discovered in this challenge binary. `CGC_Symbol_Viewer_CSV` contains a crash in the `cgc_cgcf_parse_section_header` function, which calls `memmove` with an out of bounds pointer. This crash is the only crash discovered during the unmodified fuzzer run known in the ground truth for CGC, and is noted in the documentation for the dataset.

Using modified AFL++ with Reface, over 400 crashes are discovered in `Parking Permit`, the majority of which derive from a patch to the `cgc_memcmp` function, which allows the program to artificially proceed down control flow paths it otherwise could not. For example, the majority of crashes occur during an invalid `free` which is not discovered in the original binary. This crash can be triggered by a patch that does *not* copy any extraneous data or cause memory corruption, simply returning zero when non-equal data is passed in. This example showcases potential useful properties of REFACE for deep vulnerability scanning. Although in this case we are confident this vulnerability is unreachable in the original program due to the guard comparison, code reuse is rampant within the software industry and it is entirely possible vulnerable code that was once hidden behind such a guard comparison may be used in a way that it is left unguarded in the future. `CGC_Symbol_Viewer_CSV` does not discover any additional crashes with the benefit of REFACE, however the case study below highlights its speed and coverage gain as compared to unmodified AFL++ in exploring behind a more complex guard check. `LSIMP` also does not discover additional crashes. It does discover inputs which AFL++ marks as crashes sooner, however this is not necessarily a benefit and simply reflects AFL++'s eager approach to finding what it interprets as bugs. Finally, `PKK_Steganography` experiences a problem where additional fuzzers cannot start due to discovered seeds which cause the fuzzer to hang or crash, and is unable to start.

Overall, we show that AFL++ with reface performs at least as well as unmodified AFL++ at finding crashes (1 real crash vs 1 real crash) and potentially better, although the additional discovered vulnerability is not reproducible on the original binary due to a guard check. Some results reflect areas where the system could be improved, however as shown

above the demonstrated coverage example is a strong positive outcome. We leave further attention to improving REFACE specifically for bug finding case to future work, with some discussion of avenues for improvement in Chapter 4.

3.3.4 Case Study

The `CGC_Symbol_Viewer_CSV` binary is a prototypical case study. It is a parser for CGC binary programs that parses a format similar to a simplified ELF file. The program's main function is given in Listing 13, and reads and then parses data.

The last two calls, to `cgc_cgcf_parse_file_header` and `cgc_cgcf_is_valid` are both prototypical impeding functions that take user input and perform checks on the data. In particular, the file header parse function peforms both checks *and* copying of buffer data.

The code for the parse file header function is given in Listing 14, and is conceptually simple, however the check of the buffer length depends on a previous read from user input to obtain the buffer length value. Therefore, it is difficult for the fuzzer to bypass this check as a significant weak point of fuzzing without symbolic execution is the absence of state representation, which would allow correlation of input data to the data checked. REFACE analyzes this program and identifies `cgc_cgcf_parse_file_header` as an impeding function because it receives user input via an argument (in this case, `buf`), returns skewed return values (in this case, -1 and 0, with -1 significantly more common). In addition, the `if` check on the return value at line 24 causes a stuck point. Next, REFACE attempts to derive a patch for this function automatically.

The input and output buffers (buf and hdr respectively) are detected via taint snapshotting at entry and return from the function, which allows REFACE to correctly identify the location of the buffer relative to the argument registers. In this case, `rdx` is a direct pointer to the output data, so REFACE uses this information in conjunction with analysis of return values to derive the patch given in Listing 15. The call to the original function in `main`

```
int main() {
    /* ................. */
    /* Read in size */
    if (cgc_read_n(STDIN, (char *)&size, sizeof(size)) != sizeof(size))
    return -1;

    /* Check size */
    if (size > MAX_FILE_SIZE)
    {
    cgc_printf("Too big.\n");
    return -1;
    }

    /* Allocate memory */
    file = cgc_malloc(size);
    if (file == NULL)
    return -1;

    /* Read in file */
    if (cgc_read_n(STDIN, file, size) != size)
    goto error;

    /* Parse file header */
    if (cgc_cgcf_parse_file_header(file, size, &ehdr) != 0)
    {
    cgc_printf("Invalid CGC file header.\n");
    goto error;
    }

    /* Validate CGC magic */
    if (!cgc_cgcf_is_valid(&ehdr))
    {
    cgc_printf("Invalid CGC magic.\n");
    goto error;
    }
```

Listing 13: The main function of `CGC_Symbol_Viewer_CSV`

is redirected to this function, and this function is inserted into the binary and a fuzzer is started. This function serves as a nearly optimal patch candidate, as not only does returning a new magic value (0) cause execution to continue past the error check where it previously failed, but the replacement of a copy with a read from fuzzer data allows the fuzzer to exer-

```c
int cgc_cgcf_parse_file_header(const char *buf, cgc_size_t buf_len, cgcf_Ehdr *hdr)
{
  /* Check if the pointers are NULL */
  if (buf == NULL || hdr == NULL)
    return -1;

  /* Check if the buffer is large enough */
  if (buf_len < sizeof(cgcf_Ehdr))
    return -1;

  /* Copy over the file header */
  cgc_memmove(hdr, buf, sizeof(cgcf_Ehdr));

  return 0;
}
```

Listing 14: An impeding function from `CGC_Symbol_Viewer_CSV`

```c
void read_fuzzer(uint8_t *buf, size_t size) {
    ssize_t rv = _syscall3(SYS_read, 0, (size_t)buf, size);
    if (!rv) {
        _syscall1(SYS_exit, 0);
    }
}

uint64_t patch(uint64_t arg0, uint64_t arg1, uint64_t arg2, uint64_t arg3,
    uint64_t arg4, uint64_t arg5) {

    uint8_t *outb2 = (uint8_t*)arg2;
    read_fuzzer(outb2, 8);


    uint64_t ret = 0;
    uint8_t idx = 0;
    read_fuzzer((uint8_t *)&idx, sizeof(uint8_t));
    idx %= 2;
    const uint64_t rvs[2] = {0x0, 0xffffffff};
    ret = rvs[idx];
    return ret;
}
```

Listing 15: Synthesized patch for `cgc_cgcf_parse_file_header`

cise additional control flow paths after the check. This is the cause of the surprisingly large increase in basic block coverage, as un-checked fuzzer data is now passed directly to deeper program logic in the binary.

4. FUTURE WORK

4.1 Additional Evaluation

The primary future work in this area is an extended evaluation. Due to the difficulty of evaluating REFACE in a scalable fashion, as well as the computational requirements (REFACE used 100% of 112 cores in its evaluation over the three binaries discussed previously), this is left for future work. To fully and completely evaluate the system, there are several key recommendations. First, a significantly larger dataset with many various types of programs should be used, including but not limited to the full CGC and Magma datasets. Replicating this analysis over a larger dataset will serve two purposes. First, it will provide a more accurate evaluation of the effectiveness of this approach on a wider variety of programs with different functions. The evaluation contained in this work is limited in scope and intended solely to prove the method and approach is effective, and was not intended as a comparison against other fuzzing approaches and methodologies. Second, analysis of many different types of applications may reveal other effective applications of the approach.

Second, this evaluation does not seek to discover novel bugs or exploits in programs, but future work should be done to determine the effectiveness of REFACE at discovering bugs and vulnerabilities. Here we focus only on the enhancement of coverage by the tool, but the core of the project aims to enhance the bug finding process as well. Analysis of the full CGC dataset as well as vulnerability-focused tests – such as Magma and Lava – will provide useful data regarding the power of this method for impedance abstraction for quickly discovering bugs

4.2 Buffer Analysis

The largest gap in the implementation of REFACE lies in the analysis of impeding functions to identify buffer information and inform the patcher where additional input data should be read. The primary barrier in this area is simple: if a function is executed with

user input as some argument, but the function returns prior to performing reads or writes from or to the tainted input data. This causes a problem in patch synthesis, because if a read or write operation does not occur during any trace of the program, it is effectively hidden from the analysis. we mitigate this problem with several heuristics, but there is further work in this area. Notably, static symbolic analysis of the impeding function could be used to reveal the *possible* operations, or under-constrained symbolic execution could be leveraged to solve for the real set of operations on the tainted memory under analysis. Furthermore, identification of tainted data operates recursively and therefore supports analysis nested structures and other complex data structures. However, REFACE currently makes no attempt to infer the real data type of underlying data. Heuristics could easily be leveraged to perform a best-effort analysis of the input arguments for this purpose.

4.3 De-Simplification

REFACE is designed to modify binaries and fuzz those modified binaries to potentially uncover vulnerabilities. However, it does not currently attempt to *re-host* any discovered inputs for bugs discovered in a variant binary back to the original binary. In general, we propose this step is not required – a human analyst can easily determine given a crashing input for a patched binary whether the crash is problematic and necessitates a fix, or whether it is unreachable in the real system. For a production system, however, this would be a helpful feature for scalability and triaging. The current theory for *de-simplifying* inputs is to simply fall back to dynamic symbolic execution on the original binary using the modified input, fixing up constraints to coerce the program to follow the same path as the variant binary. However, this approach is not foolproof and relies on pure symbolic execution, so additional research into its viability or alternative methods is recommended.

There is ample future work in fuzzing, and particularly in the direction REFACE aims to proceed. Abstraction of program elements, including but not limited to functions, is a promising direction for many reasons discussed prior. Though this work highlights only a

modest increase in fuzzing performance, the promise of additional bugs given investment in engineering and heuristics for improved patch generation is motivation for further investigation.

5. CONCLUSIONS

5.1 Contributions

This work makes several material contributions to the practice of fuzzing and scalable analysis. First, we put forth REFACE, a pipeline for integrating function-level fuzzer impedance analysis into a normal fuzzing process composed of a scalable dynamic tracing framework based on QEMU, targeted taint analysis tools based on `libdft64`, and patching tools built with the LIEF binary patching framework. The core REFACE implementation consists of approximately 3000 lines of Rust, 2000 lines of C++, and 1000 lines of Python, with the following sub-components: cannonball tracing framework (1000 lines of Rust code and 1200 lines of C), pypatches patching framework (1500 lines of Python), pysquishy shellcode compiler (500 lines of C++ and 800 lines of Python). In addition, we put forth a general method for analysis of functions for control flow impedance and a limited analysis of results of the application of this general method with positive results on appropriate targets.

We demonstrate a confirmation of the hypothesis that abstraction of stuck points to impeding functions and modification to affect control flow increases coverage, and we demonstrate potential to confirm the hypothesis with respect to bugs found, leaving a wider-scale evaluation to future work. In addition, we evaluate the fitness of REFACE as a component of a wider fuzzing research landscape and identify similar works.

5.2 Takeaways

Despite a lack of clear bug discovery, we suspect primarily due to a small-scale evaluation, several interesting observations arise through analysis of the various stages of results:

- Near 1:1 correlation of stuck points and coverage bears investigation, it is possible coverage may be used as a metric of impedance as well as discovery.
- Sharp decreases in coverage and new input discovery over time should be evaluated, and solutions may present avenues for investigation.

- Bugs discovered may be true positives and remain unreachable in *current code*.

Most critically, we consider investigation of methods for abstraction of concepts that present difficulty to fuzzers a fruitful research area. Intuitively, reducing the complexity of specific areas of binary programs to allow fuzzers to focus on other areas of a binary should allow greater exploration of control and data flows. Abstraction of difficult to bypass checks from a statement level to a function level is only one application of an abstraction-based approach to fuzzing improvement. For example, in very large programs the abstraction of entire components into opaque data producing or consumers could provide an even higher-level analog of this idea. However, many difficulties remain unsolved. Chief among them is the problem of unsoundness when attempting to generalize software defects identified in modified binaries back to the original binary. This "re-hosting" problem is a significant hurdle for this line of research, and should be investigated thoroughly.

Empirically, we present data describing the initial validity of the approach outlined in the system design. REFACE discovers 75% of first-encountered stuck points across four binaries. REFACE discovered a minimum of 2 and maximum of 5 impeding functions over a 24 hour fuzzing execution and was able to generate variant binaries and begin fuzzing these variants on 3 out of 4 binaries. Fuzzing of variant binaries generated through modification of discovered impeding functions showed a significant increase in coverage in the same time interval when compared to the original binary. Finally, REFACE discovered one new crash in a variant binary, although this new crash is unreachable in the original binary. Several avenues to improve REFACE were identified in Chapter 4. REFACE implements a proof of concept that validates the hypothesis, but falls short of demonstrating significant capability to discover new bugs on a wide variety of programs. This deficiency is primarily due to insufficient analysis of program data buffers, which we are confident implementation of ideas set forth in the Chapter 4 will address.

REFERENCES

[1] B. C. HOUSEL, "A STUDY OF DECOMPILING MACHINE LANGUAGES INTO HIGH-LEVEL MACHINE INDEPENDENT LANGUAGES," *Theses and Dissertations Available from ProQuest*, pp. 1–271, Jan. 1, 1973. [Online]. Available: https://docs.lib.purdue.edu/dissertations/AAI7404980.

[2] R. Farrow, K. Kennedy, and L. Zucconi, "Graph grammars and global program data flow analysis," in *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, ISSN: 0272-5428, Oct. 1976, pp. 42–56. DOI: 10.1109/SFCS.1976.17.

[3] W. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 266–278, Jul. 1977, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.1977.231144.

[4] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/360051.360056. [Online]. Available: https://dl.acm.org/doi/10.1145/360051.360056 (visited on 10/05/2022).

[5] S. K. Robinson and I. S. Torsun, "An empirical analysis of FORTRAN programs," *The Computer Journal*, vol. 19, no. 1, pp. 56–62, Jan. 1, 1976, ISSN: 0010-4620. DOI: 10.1093/comjnl/19.1.56. [Online]. Available: https://doi.org/10.1093/comjnl/19.1.56 (visited on 09/08/2022).

[6] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '77, New York, NY, USA: Association for Computing Machinery, Jan. 1, 1977, pp. 238–252, ISBN: 978-1-4503-7350-0. DOI: 10.1145/512950.512973. [Online]. Available: https://doi.org/10.1145/512950.512973 (visited on 09/08/2022).

[7] (). "The LLVM compiler infrastructure project," [Online]. Available: https://llvm.org/ (visited on 10/24/2022).

[8] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA: IEEE, May 2012, pp. 380–394, ISBN: 978-1-4673-1244-8 978-0-7695-4681-0. DOI: 10.1109/SP.2012.31. [Online]. Available: https://ieeexplore.ieee.org/document/6234425/ (visited on 09/24/2022).

[9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2011, pp. 463–469, ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_37.

[10] C. Cifuentes, "Reverse compilation techniques," 1994.

[11] C. Cifuentes and A. Fraboulet, "Intraprocedural static slicing of binary executables," in *In Int. Conf. on Softw. Maint*, 1997, pp. 188–195.

[12] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, Dec. 1, 1992, ISSN: 1057-4514. DOI: 10.1145/161494.161501. [Online]. Available: https://doi.org/10.1145/161494.161501 (visited on 09/08/2022).

[13] H. Theiling, "Extracting safe and precise control flow from binaries," in *In Proc. 7th Conference on Real-Time Computing Systems and Applications*, 2000.

[14] M. Egele, T. Scholte, E. Kirda, and C. Krügel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, 6:1–6:42, 2008.

[15] S. Shankland. (). "VMware ready to capitalize on hot server market," CNET, [Online]. Available: https://www.cnet.com/tech/tech-industry/vmware-ready-to-capitalize-on-hot-server-market/ (visited on 10/24/2022).

[16] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," *Science of Computer Programming*, Special Issue on Program Comprehension, vol. 40, no. 2, pp. 171–188, Jul. 1, 2001, ISSN: 0167-6423. DOI: 10.1016/S0167-6423(01)00014-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642301000144 (visited on 09/08/2022).

[17] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy, "Interprocedural static slicing of binary executables," in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, Sep. 2003, pp. 118–127. DOI: 10.1109/SCAM.2003.1238038.

[18] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 63–68, Dec. 2005, ISSN: 0163-5964. DOI: 10.1145/1127577.1127590. [Online]. Available: https://dl.acm.org/doi/10.1145/1127577.1127590 (visited on 09/08/2022).

[19] (2007). "Decompilation gets real – hex rays," [Online]. Available: https://hex-rays.com/blog/decompilation-gets-real/ (visited on 09/08/2022).

[20] (2004). "Static disassembly of obfuscated binaries," [Online]. Available: https://www.usenix.org/legacy/event/sec04/tech/full_papers/kruegel/kruegel_html/ (visited on 09/08/2022).

[21] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, ISSN: 1063-9527, Miami Beach, FL, USA: IEEE, Dec. 2006, pp. 269–278, ISBN: 978-0-7695-2716-1. DOI: 10.1109/ACSAC.2006.50. [Online]. Available: http://ieeexplore.ieee.org/document/4041173/ (visited on 09/08/2022).

[22] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Information Systems Security*, R. Sekar and A. K. Pujari, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2008, pp. 1–25, ISBN: 978-3-540-89862-7. DOI: 10.1007/978-3-540-89862-7_1.

[23] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Computer Aided Verification*, A. Gupta and S. Malik, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2008, pp. 423–427, ISBN: 978-3-540-70545-1. DOI: 10.1007/978-3-540-70545-1_40.

[24] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "CodeSurfer/x86—a platform for analyzing x86 executables," in *Compiler Construction*, R. Bodik, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2005, pp. 250–254, ISBN: 978-3-540-31985-6. DOI: 10.1007/978-3-540-31985-6_19.

[25] (). "Cyber grand challenge," [Online]. Available: https://www.darpa.mil/program/cyber-grand-challenge (visited on 10/24/2022).

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*, Austin, Texas, United States: ACM Press, 1989, pp. 25–35, ISBN: 978-0-89791-294-5. DOI: 10.1145/75277.75280. [Online]. Available: http://portal.acm.org/citation.cfm?doid=75277.75280 (visited on 09/09/2022).

[27] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, "Simple and efficient construction of static single assignment form," p. 20,

[28] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, *Code2vec: Learning distributed representations of code*, Oct. 30, 2018. DOI: 10.48550/arXiv.1803.09473. arXiv: 1803.09473[cs,stat]. [Online]. Available: http://arxiv.org/abs/1803.09473 (visited on 09/09/2022).

[29] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, *GraphCodeBERT: Pre-training code representations with data flow*, Sep. 13, 2021. arXiv: 2009.08366[cs]. [Online]. Available: http://arxiv.org/abs/2009.08366 (visited on 10/24/2022).

[30] P. Biondi, R. Rigo, S. Zennou, and X. Mehrenberger, "BinCAT: Purrfecting binary static analysis," p. 22,

[31] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," p. 12,

[32] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/360248.360252. [Online]. Available: https://dl.acm.org/doi/10.1145/360248.360252 (visited on 09/24/2022).

[33] S. L. Hantler and J. C. King, "An introduction to proving the correctness of programs," *ACM Computing Surveys*, vol. 8, no. 3, pp. 331–353, Sep. 1976, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/356674.356677. [Online]. Available: https://dl.acm.org/doi/10.1145/356674.356677 (visited on 09/18/2022).

[34] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, and G. Weikum, vol. 4963, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78799-0 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24. [Online]. Available: http://link.springer.com/10.1007/978-3-540-78800-3_24 (visited on 09/24/2022).

[35] J. C. King, "A new approach to program testing," p. 13, 1974.

[36] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT—a formal system for testing and debugging programs by symbolic execution," *ACM SIGPLAN Notices*, vol. 10, no. 6, pp. 234–245, Jun. 1975, ISSN: 0362-1340, 1558-1160. DOI: 10.1145/390016.808445. [Online]. Available: https://dl.acm.org/doi/10.1145/390016.808445 (visited on 09/18/2022).

[37] (). "Abstract interpretation in a nutshell," [Online]. Available: https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html (visited on 09/25/2022).

[38] J. Darringer and J. King, "Applications of symbolic execution to program testing," *Computer*, vol. 11, no. 4, pp. 51–60, Apr. 1978, Conference Name: Computer, ISSN: 1558-0814. DOI: 10.1109/C-M.1978.218139.

[39] T. Ball, "Abstraction-guided test generation: A case study," p. 16, 2003.

[40] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," p. 19, 2004.

[41] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for c," p. 10, 2005.

[42] Y. Xie and A. Aiken, "Scalable error detection using boolean satisfiability," p. 13, 2005.

[43] D. Engler and D. Dunbar, "Under-constrained execution: Making automatic code destruction easy and scalable," in *Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA '07*, London, United Kingdom: ACM Press, 2007, pp. 1–4, ISBN: 978-1-59593-734-6. DOI: 10.1145/1273463.1273464. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1273463.1273464 (visited on 09/24/2022).

[44] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," p. 17,

[45] STP. (). "The simple theorem prover," The Simple Theorem Prover, [Online]. Available: https://stp.github.io// (visited on 10/24/2022).

[46] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," p. 13, 2008.

[47] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Transactions on Computer Systems*, vol. 30, no. 1, pp. 1–49, Feb. 2012, ISSN: 0734-2071, 1557-7333. DOI: 10.1145/2110356.2110358. [Online]. Available: https://dl.acm.org/doi/10.1145/2110356.2110358 (visited on 09/24/2022).

[48] F. Bellard, "QEMU, a fast and portable dynamic translator," p. 6, 2005.

[49] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA: IEEE, May 2016, pp. 138–157, ISBN: 978-1-5090-0824-7. DOI: 10.1109/SP.2016.17. [Online]. Available: http://ieeexplore.ieee.org/document/7546500/ (visited on 09/24/2022).

[50] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2016, ISBN: 978-1-891562-41-9. DOI: 10.14722/ndss.2016.23368. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf (visited on 09/14/2022).

[51] A. Vishnyakov, A. Fedotov, D. Kuts, A. Novikov, D. Parygina, E. Kobrin, V. Logunova, P. Belecky, and S. Kurmangaleev, "Sydr: Cutting edge dynamic symbolic execution," in *2020 Ivannikov Ispras Open Conference (ISPRAS)*, Dec. 2020, pp. 46–54. DOI: 10.1109/ISPRAS51486.2020.00014.

[52] S. Poeplau and A. Francillon, "SymQEMU: Compilation-based symbolic execution for binaries," in *Proceedings 2021 Network and Distributed System Security Symposium*, Virtual: Internet Society, 2021, ISBN: 978-1-891562-66-2. DOI: 10.14722/ndss.2021.24118. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_2B-2_24118_paper.pdf (visited on 08/29/2022).

[53] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: A powerful approach to weakest preconditions," p. 12, 2009.

[54] A. Brucato, "Semi-automated identification and handling of input parsing routines for efficient fuzzing and symbolic execution," pp. 1–124, 2019.

[55] N. Nethercote and J. Seward. (). "Valgrind: A program supervision framework," [Online]. Available: https://reader.elsevier.com/reader/sd/pii/S1571066104810429?token=CEBAE7D1B88CE7690C2313AE06D085E80DC02C7824FF0FF37C8978CEE1FF8785B0541A2A943D295771963A173DC85538&originRegion=us-east-1&originCreation=20221003032231 (visited on 10/03/2022).

[56] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA '07*, London, United Kingdom: ACM Press, 2007, p. 196, ISBN: 978-1-59593-734-6. DOI: 10.1145/1273463.1273490. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1273463.1273490 (visited on 10/05/2022).

[57] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant, "DECAF: A platform-neutral whole-system dynamic binary analysis platform," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 164–184, Feb. 2017, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.2016.2589242.

[58] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," p. 15,

[59] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, ISSN: 2375-1207, May 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046.

[60] L. R. L. Saiider, "A GENERAL TEST DATA GENERATOR FOR COBOL," p. 8, 1962.

[61] C. Burgess, "Software TEsting using an automatic generator of test data," 1970. [Online]. Available: https://web.archive.org/web/20170814021946/https://www.witpress.com/Secure/elibrary/papers/SQM93/SQM93040FU.pdf (visited on 09/15/2022).

[62] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 156–173, Jun. 1975, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.1975.6312836.

[63] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1, 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279. [Online]. Available: https://doi.org/10.1145/96267.96279 (visited on 09/14/2022).

[64] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services," p. 23, 1995.

[65] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows NT applications using random testing," p. 10, 2000.

[66] D. Aitel. (2002). "The advantages of block-based protocol analysis for security testing," [Online]. Available: https://www.immunitysec.com/downloads/advantages_of_block_based_analysis.html (visited on 09/15/2022).

[67] *MozillaSecurity/peach*, original-date: 2015-06-15T20:58:45Z, Aug. 28, 2022. [Online]. Available: https://github.com/MozillaSecurity/peach (visited on 08/29/2022).

[68] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA: IEEE, May 2017, pp. 579–594, ISBN: 978-1-5090-5533-3. DOI: 10.1109/SP.2017.23. [Online]. Available: http://ieeexplore.ieee.org/document/7958599/ (visited on 09/14/2022).

[69] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for deep bugs with grammars," in *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2019, ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23412. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_04A-3_Aschermann_paper.pdf (visited on 09/14/2022).

[70] J. Wang, B. Chen, L. Wei, and Y. Liu, *Superion: Grammar-aware greybox fuzzing*, Jan. 23, 2019. arXiv: 1812.01197[cs]. [Online]. Available: http://arxiv.org/abs/1812.01197 (visited on 09/14/2022).

[71] H. Bos, "Vrije universiteit amsterdam, the netherlands {asia,tsu500,herbertb}@few.vu.nl," p. 20,

[72] C. Miller and Z. N. J. Peterson, "Analysis of mutation and generation-based fuzzing," p. 7, 2007.

[73] M. Zalewski, *Bunny the fuzzer*, original-date: 2022-09-17T17:22:52Z, Sep. 17, 2022. [Online]. Available: https://github.com/novafacing/bunny/blob/1001d7dc44a765ca0632c169c462494d3ea4d097/README (visited on 09/17/2022).

[74] M. IvankoviÄ‡, G. PetroviÄ‡, R. Just, and G. Fraser, "Code coverage at google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn Estonia: ACM, Aug. 12, 2019, pp. 955–963, ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3340459. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3340459 (visited on 10/24/2022).

[75] M. Last, S. Eyal, and A. Kandel, "Effective black-box testing with genetic algorithms," in *Hardware and Software, Verification and Testing*, S. Ur, E. Bin, and Y. Wolfsthal, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, and G. Weikum, vol. 3875, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 134–148, ISBN: 978-3-540-32604-5 978-3-540-32605-2. DOI: 10.1007/11678779_10. [Online]. Available: http://link.springer.com/10.1007/11678779_10 (visited on 09/18/2022).

[76] A. Zeller. (). "Search-based fuzzing - the fuzzing book," [Online]. Available: https://www.fuzzingbook.org/html/SearchBasedFuzzer.html (visited on 09/18/2022).

[77] M. Zalewski. (2013). "AFL history," [Online]. Available: https://lcamtuf.coredump.cx/afl/historical_notes.txt (visited on 09/15/2022).

[78] (). "Lcamtuf's blog: Binary fuzzing strategies: What works, what doesn't," [Online]. Available: https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html (visited on 09/18/2022).

[79] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2017, ISBN: 978-1-891562-46-4. DOI: 10.14722/ndss.2017.23404. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/ (visited on 09/14/2022).

[80] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Montpellier France: ACM, Sep. 3, 2018, pp. 475–485, ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238176. [Online]. Available: https://dl.acm.org/doi/10.1145/3238147.3238176 (visited on 09/14/2022).

[81] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2019, ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23371. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_04A-2_Aschermann_paper.pdf (visited on 09/14/2022).

[82] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," p. 12, 2020.

[83] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Event USA: ACM, Jul. 18, 2020, pp. 1–13, ISBN: 978-1-4503-8008-9. DOI: 10.1145/3395363.3397372. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397372 (visited on 09/18/2022).

[84] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, and Y. Song, "MOPT: Optimized mutation scheduling for fuzzers," p. 19,

[85] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the 44th International Conference on Software Engineering*, Pittsburgh Pennsylvania: ACM, May 21, 2022, pp. 1634–1645, ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510174. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510174 (visited on 09/14/2022).

[86] K. Sen, "DART: Directed automated random testing," p. 1, 2005.

[87] W. Drewry and T. Ormandy. (2007). "Flayer: Exposing application internals USENIX," [Online]. Available: https://www.usenix.org/conference/woot-07/flayer-exposing-application-internals (visited on 09/15/2022).

[88] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2016, ISBN: 978-1-891562-41-9. DOI: 10.14722/ndss.2016.23368. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf (visited on 08/29/2022).

[89] S. Poeplau and A. Francillon, "Symbolic execution with SYMCC: Don't interpret, compile!," p. 19, 2020.

[90] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, Taowei, and L. Lu, *SAVIOR: Towards bug-driven hybrid testing*, Jun. 17, 2019. arXiv: 1906.07327[cs]. [Online]. Available: http://arxiv.org/abs/1906.07327 (visited on 09/14/2022).

[91] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA: IEEE, May 2018, pp. 697–710, ISBN: 978-1-5386-4353-2. DOI: 10.1109/SP.2018.00056. [Online]. Available: https://ieeexplore.ieee.org/document/8418632/ (visited on 09/14/2022).

[92] X. Liu, Q. Wei, Q. Wang, Z. Zhao, and Z. Yin, "CAFA: A checksum-aware fuzzing assistant tool for coverage improvement," *Security and Communication Networks*, vol. 2018, pp. 1–13, Oct. 16, 2018, ISSN: 1939-0114, 1939-0122. DOI: 10.1155/2018/9071065. [Online]. Available: https://www.hindawi.com/journals/scn/2018/9071065/ (visited on 10/05/2022).

[93] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," p. 18,

[94] (). "wolfSSL embedded SSL/TLS library now supporting TLS 1.3," [Online]. Available: https://www.wolfssl.com/ (visited on 10/24/2022).

[95] (). "axTLS embedded SSL," [Online]. Available: https://axtls.sourceforge.net/ (visited on 10/24/2022).

[96] (). "Openssl," [Online]. Available: https://www.openssl.org/ (visited on 10/24/2022).

[97] (). "3. pure functions, laziness, i/o, and monads - school of haskell school of haskell," [Online]. Available: https://web.archive.org/web/20161027145455/https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io (visited on 10/17/2022).

[98] (). "Debian sources debian sources," [Online]. Available: https://sources.debian.org/ (visited on 08/29/2022).

[99] F. Wang and Y. Shoshitaishvili, "Angr - the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, Sep. 2017, pp. 8–9. DOI: 10.1109/SecDev.2017.14.

[100] (). "Pin - a dynamic binary instrumentation tool," Intel, [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html (visited on 10/24/2022).

[101] *SheLLVM*, original-date: 2018-01-19T08:03:41Z, Oct. 11, 2022. [Online]. Available: https://github.com/SheLLVM/SheLLVM (visited on 10/24/2022).

[102] QuarksLab. (Jul. 18, 2021). "Home," LIEF, [Online]. Available: https://lief-project.github.io/ (visited on 10/24/2022).

[103] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "FuzzBench: An open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens Greece: ACM, Aug. 20, 2021, pp. 1393–1403, ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3473932. [Online]. Available: https://dl.acm.org/doi/10.1145/3468264.3473932 (visited on 08/29/2022).

[104] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, QC, Canada: IEEE, Mar. 2015, pp. 560–564, ISBN: 978-1-4799-8469-5. DOI: 10.1109/SANER.2015.7081877. [Online]. Available: http://ieeexplore.ieee.org/document/7081877/ (visited on 10/24/2022).

[105] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Hong Kong China: ACM, Nov. 11, 2014, pp. 654–665, ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635929. [Online]. Available: https://dl.acm.org/doi/10.1145/2635868.2635929 (visited on 10/24/2022).

[106] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, *Evaluating fuzz testing*, Oct. 18, 2018. arXiv: 1808.09700[cs]. [Online]. Available: http://arxiv.org/abs/1808.09700 (visited on 08/29/2022).

[107] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad India: ACM, May 31, 2014, pp. 435–445, ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568271. [Online]. Available: https://dl.acm.org/doi/10.1145/2568225.2568271 (visited on 10/24/2022).

[108] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, Pittsburgh Pennsylvania: ACM, May 21, 2022, pp. 1621–1633, ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510230. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510230 (visited on 08/29/2022).

[109] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA: IEEE, May 2016, pp. 110–121, ISBN: 978-1-5090-0824-7. DOI: 10.1109/SP.2016.15. [Online]. Available: http://ieeexplore.ieee.org/document/7546498/ (visited on 10/24/2022).

[110] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, Nov. 30, 2020, ISSN: 2476-1249. DOI: 10.1145/3428334. [Online]. Available: https://dl.acm.org/doi/10.1145/3428334 (visited on 10/24/2022).

[111] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing," p. 18,

[112] (). "Honggfuzz," honggfuzz, [Online]. Available: https://honggfuzz.dev/ (visited on 10/24/2022).

[113] (). "CodeQL," [Online]. Available: https://codeql.github.com/ (visited on 10/24/2022).