Exploring Abstraction Functions in Fuzzing

Christopher Salls*, Aravind Machiry*, Adam Doupe†,
Yan Shoshitaishvili†, Christopher Kruegel*, and Giovanni Vigna*
*University of California, Santa Barbara {salls, machiry, chris, vigna}@cs.ucsb.edu
†Arizona State University {doupe, yans}@asu.edu

***Abstract*—Fuzz testing has emerged as the preeminent automated security analysis technique in the real world. To keep up with the shifting security landscape, researchers have innovated the fuzzing process to identify more and more complex vulnerabilities. One innovation is an approach inspired by genetic programming: the fuzzer generates test-cases, *evaluates* the quality of the test-case, and uses this evaluation to *select* test-cases for further iterations of the process. While this innovation has impressive results: without a formal, scientific model on which to base these improvements, the field of fuzzing has been explored in an *ad hoc* way. As a result, it is difficult to understand the relative merit of different techniques.**

In this paper, we formalize the input evaluation and selection components of fuzzing, borrowing concepts from the field of static analysis, and providing a base for future expansion of and research into fuzzing techniques. In building this formalism, we observed that the impact of different *abstraction functions* in modern fuzzing techniques is under-explored in prior research. Without a formal base on which to reason about their contributions, researchers of fuzzing techniques have missed the potential for improvements to this critical component of fuzzing approaches. We explore the implications of our formalization-derived observation on the effectiveness of evolutionary fuzzing techniques in the second half of the paper, showing that the application of different abstraction functions, and the use of multiple abstraction functions in tandem, improves state-of-the-art fuzzing techniques.

I. Introduction

As our society becomes increasingly dependent on software, the security of this software becomes paramount. Whereas, at one time, security could be approached *reactively*—responding to vulnerabilities only after their exploitation by attackers—this is no longer acceptable. Modern security is *proactive*, with researchers attempting to identify and fix software flaws *before* they can be found by attackers.

One method, fuzz testing, has emerged as the preeminent automated security analysis technique in the real world. This technique has impressive results thus far. A modern example is American Fuzzy Lop (AFL) [23], a powerful fuzzer based on genetic techniques, which is responsible for the detection of hundreds of real-world flaws, including high-impact vulnerabilities such as the Stage-fright vulnerability [7]. AFL's success has spawned a veritable "cottage industry" of researchers looking to improve various stages of the fuzzing process. However, without a formal, scientific model on which to base these improvements, the field of fuzzing was explored in an extremely *ad hoc* way, and it is difficult to understand the relative merit of different approaches.

In this paper, we present the first formalization of input evaluation and selection in fuzzing, borrowing concepts from the field of static analysis. Formally, a fuzzer generates input test-cases and dispatches them to a program, dynamically triggering a subset of the potential states that the program can reach. However, the full set of states is potentially infinite. For tractability, the formal fuzzing process uses an approach-specific *abstraction function* to reduce this set to an *abstract state space*, allowing the fuzzer to identify "promising" test-cases for further mutation by selecting test-cases that correlate to different abstract states in the state space.

We explore the implications of our formalization-derived observation on the effectiveness of evolutionary fuzzing techniques in the second half of the paper, and we show that the application of different abstraction functions, and the use of multiple abstraction functions in tandem, shows promise for improving state-of-the-art fuzzing techniques. To maximize the benefit of this work to the community, we will open-source the resulting tool upon publication.

In summary, this paper makes the following contributions:

- To form a scientific base for research into future approaches in fuzzing, we provide a formalization of the input evaluation and selection process in fuzzing and redefine current work in the context of this formalism.
- Stemming directly from an observation made during the formalization process, we propose a diversification of fuzzer abstraction functions and design a number of such functions that can be used both alone and in composition with each other.
- We implement an extensible framework for the development and evaluation of fuzzer abstraction functions, evaluate its impact on the effectiveness of a modern fuzzer, and open-source our work for reproducibility and for the community to build upon.

II. Background and Related Work

Fuzzing is a well-known technique for program testing by generating random data as *input* to programs under test, and has drawn much research attention over a wide span of time. The main goal of fuzzing techniques is to violate implicit expectations made by the developer on the input and expose resulting security flaws or bugs.

Input generation: This defines how the inputs are generated by the fuzzing technique. Most of the research in fuzzing occurs

978-1-7281-4760-4/20/$31.00 ©2020 IEEE

in this aspect. There are many well-known input generation techniques:

Mutation-based: Here, the fuzzer starts with some seed inputs and new inputs are generated by mutating certain regions of existing inputs [23].

Evolution-based: In this case, evolutionary techniques are used to combine interesting inputs to generate new inputs [17]. We formally define the definition of *interesting* in Section III.

Grammar-based: The fuzzer generates input that satisfies a specified grammar. Program that expect input to conform to a grammar, such as interpreters, and file editors, are generally fuzzed with this input generation technique. If the input structure can be specified as a grammar, then grammar-based techniques can be effective in triggering complex behavior in the target program.

Note that these techniques are not exclusive—tools, such as Dowser [11], combine taint tracking and symbolic execution to generate interesting inputs to trigger buffer overflows. Similarly, Driller [19] combines three strategies: mutation, evolution, and symbolic execution.

Input selection or evaluation: A fuzzing technique must know the effectiveness of its generated inputs, so that it can determine if a new input or mutation strategy was useful. Measuring the effectiveness needs visibility into the program under test and, as such, black-box techniques can only have a limited input evaluation by examining the output of the program. In Section III we formally redefine input selection and input evaluation as abstraction functions, however for now we will use the informal terms. There are two well-known input evaluation techniques:

Goal-based or directed: Here the input is evaluated on the likelihood of achieving a goal or causing program to reach a certain state. Dowser [11] and BuzzFuzz [8] generate inputs that are likely to cause buffer overflows. libFuzzer-gv [20] explored guiding the fuzzer based on the stack depth and on the number of memory allocations made. However, these input selection strategies are specialized for each goal and cannot be generalized.

Coverage-based: Here the input is evaluated based on what code is triggered by it on the test program. The intuition is simple: *dynamic techniques cannot find a bug if they do not execute the code containing the bug*, and thus, a higher code coverage implies a higher chance of bugs. Most general fuzzing techniques, such as AFL [23], VUzzer [17], and syzkaller [10] are coverage based.

Most existing research in fuzzing attempts to find new input generation techniques to efficiently generate effective inputs. However, the importance of input selection and evaluation in fuzzing remains an under-explored area and a promising research direction. Wang et al [22] take a step down that road, exploring the differences in some selection strategies. Our work (done independently and concurrently) further explores the effectiveness of various abstraction functions, evaluating a different set of fuzzing strategies, in a larger experiment, with a thorough investigation into the effectiveness of the various strategies.

III. Formalizing

Fundamentally, the goal of a fuzzer is to find in a given program software bugs that violate the security properties of the program. As fuzzing is a dynamic technique, the fuzzer finds bugs by providing input to the program in an attempt to trigger a program state that violates a security specification (for example, accessing invalid memory). Fuzzing can be viewed as an iterative process, targeted to explore the state space of a given program *completely*. Unfortunately, exploring the entire state space of a program is equivalent to solving the halting problem, which is undecidable.

Similar to program testing, fuzzing is an automated testing technique. It tries to generate *interesting* inputs as fast as possible within a given resource budget.

A. Concepts

First, we define some notions that will be used throughout our fuzzing formalism:

Input: Programs consume input data to drive their operation. An *input*, ι is our representation of this input data. Note that this notion of input covers all types of input to a program (command line, network, files etc).

Input Space: The alphabet of possible inputs is Σ and the set of all possible inputs is Σ^*. If the length of the input is not bounded, the set Σ^* is infinite.

Concrete State: The snapshot of all the processor registers, the program's memory, file system operations, or anything else that effects the operation of the program represents the concrete state, r, of a program. The symbol C indicates the set of all the possible concrete states of a program.

Concrete State Space: Each input ι triggers a series of concrete states as it is processed. The trace of all of the concrete states reached by the input ι, denoted as cs_ι, is the Concrete State Trace of the input.
Because there is a potentially infinite amount of inputs that could be read by the program, the set of all Concrete State Traces can be infinite. CS denotes the set of all concrete state traces for a program.

Abstract State Space, AS: As the set of Concrete State Traces CS can be infinite (or computationally infeasible to enumerate), fuzzing techniques must abstract the concrete state space so that different states (and therefore, different inputs) can be considered equivalent. Here we derive inspiration from Abstract Interpretation [4], and similar to abstract domains in Abstract Interpretation we define the *Abstract State Space* (AS) as a domain to which a concrete state space will be mapped to. The elements of this domain are called *abstract states*.

These concepts will be used as the basis of a formal definition of fuzzing.

B. Mapping to Abstract States

Fuzzing techniques reason over the abstract state space of an input instead of the concrete state space. This allows them to group inputs having different cs enabling efficient generation of interesting inputs.

The mapping between AS and CS is handled by two functions:

Abstraction function (α): This function maps a concrete state trace to an abstract state.
Formally, $\alpha : CS \rightarrow AS$ and A is a set of abstraction functions.

Concretization function (γ): This function maps an abstract state to a list of concrete states.
Formally, $\gamma : AS \rightarrow CS$.

For a given input ι and a corresponding cs_ι, we can compute the corresponding abstract state by applying α as $\alpha(cs_\iota)$. We call this the *Input Abstract State (IAS)*.

Formally, $IAS(\iota, \alpha) = \alpha(cs_\iota)$.

The tuple of an input (ι) and corresponding IAS form the *Fuzzing Result* (FR) of the input ι.

Formally, $FR(\iota, \alpha) = (\iota, IAS(\iota, \alpha))$.

For a set of inputs I and an abstraction function α, the set of corresponding fuzzing results are called *Fuzzing Results Set* (FRS).

Formally, $FRS(I, \alpha) = \{FR(\iota, \alpha), .. \mid \forall \iota \in I\}$.

For a set of inputs I and a set of abstraction functions A, we can define **Complete Fuzzing Results** (CFR) formally as:

$CFR(I, A) = \{FR(\iota, \alpha), .. \mid \forall \alpha \in A, \iota \in I\}$.

For a given set of abstraction functions A, two inputs ι_1 and ι_2 are considered the same iff $CFR(\{\iota_1\}, A) = CFR(\{\iota_2\}, A)$.

To shorten the notation we will use α (ι) as the abstraction of the Concrete State Trace triggered by ι, i.e. $\alpha(\iota) := \alpha(cs_\iota)$.

C. Fuzzing Techniques and Procedures

With these notions defined, we can formally define a fuzzing technique ($\hat{F}$) as a function that takes the following inputs:

- The set of abstraction functions to be used for the current iteration (A_{curr}).
- The program p_{curr} to be tested, with additional instrumentation as needed by the abstraction functions A_{curr}.
- The set of complete fuzzing results of all previously tested inputs and abstraction functions (CFR_{prev}).
- A set of inputs to be used for current iteration (I_{curr}).
- The time and resource consumed thus far (t r_{curr}).

and produces the following outputs:

- A set of inputs to be used for the next iteration (I_{next}).
- A set of abstraction functions to be used for the next iteration (A_{next}).
- A version of the program p_{next}, with additional instrumentation needed by the abstraction functions A_{next}.
- The new complete fuzzing result set, which includes the complete fuzzing result of I_{curr}. i.e., $\{CFR_{prev} \cup CFR(I_{curr}, A_{curr})\}$.
- The new time and resource consumption (t r_{next}).

Formally, a fuzzing technique can be defined as:

$$\hat{F} : (p_{all}, I_{all}, P(A), P(CFR_{all}), t_{all}, r_{all})$$
$$X \ (p_{all}, I_{all}, P(A), P(CFR_{all}), t_{all}, r_{all})$$

where p_{all} is the set of all possible functionally identical copies of the program p, $P(A)$ is the power set of all possible abstraction functions, $P(CFR_{all})$ is the power set of complete fuzzing results across all possible inputs I_{all} and all possible sets of abstraction functions A (formally, $CFR_{all} = CFR(I_{all}, A)$), t_{all} is the set of all possible time consumptions, and r_{all} is the set of all possible resource consumptions.

In every iteration, a fuzzing procedure usually stores all the interesting inputs used during the iteration. An input is *interesting* if it explored an abstract state that is *not* reached by any of the previous inputs. Formally, let the set of all interesting inputs stored by a fuzzing technique be $\widetilde{I}$, an input ι in the iteration, i.e., $\iota \in I_{curr}$, can be considered *interesting* if the following relation holds:

$$\exists \alpha \in A_{curr} \mid \alpha(\iota) \not\subseteq_\alpha \bigcup_{i \in \widetilde{I}} \alpha(i)$$

where A_{curr} is the set of abstraction functions used in the iteration.

The above relation ensures that there is an abstraction function, according to which the input ι explored an abstract state that is not reached by any of the previously used inputs.

The *fuzzing process* $\bar{F}$ applies the fuzzing technique $\hat{F}$ iteratively until the resource budgets are consumed.

A fuzzing process $\bar{F}$ is given the program p, an initial set of inputs I_{init} (known colloquially as "seeds"), a set of initial abstraction functions A_{init}, a fuzzing technique $\hat{F}$, a time budget t, and a resource budget r. A fuzzing process will return the set inputs that trigger invalid program states $I_{invalid}$.

$$\bar{F} : (p_{all}, I_{all}, \hat{F}_{all}, t_{all}, r_{all}) \ X \ I_{all} \tag{1}$$

The fuzzing process $\bar{F}$ applies $\hat{F}$ iteratively until the provided resource budgets are exhausted.

$$... \ \hat{F} \circ \hat{F} \circ \hat{F} \circ \hat{F}(p, I_{init}, A_{init}, \emptyset, 0, 0) \tag{2}$$

D. Fuzzing parameters

Given this formal definition of fuzzing, *any* fuzzing technique can be described by defining the following parameters:

Input Generation (I_{next}): This describes how the new input is generated, i.e., the technique used to generate I_{next} for an iteration of the fuzzing technique. An intelligent input generation is an innate feature of any fuzzing technique. There are different ways to generate inputs, whether by using random data, by using the fuzzing result of the previous inputs, or a combination of both.

Abstraction Functions (A): This is the total set of abstraction functions that could be used by the fuzzing process.

Abstraction Function Selection (A_{next}): Similar to input generation, this parameter describes the mechanism used to select the abstraction functions to be used for the next iteration. Most of the fuzzing techniques have a single abstraction function (i.e., $|A| = 1$) and use the same abstraction functions in every iteration (i.e., $A_{next} = A_{prev}$). We call these techniques *Single Abstraction Fuzzing (SAF)*.

To demonstrate the generality of our model, let us define some of the existing fuzzing technique using these parameters.

AFL [23]: For input generation, AFL uses various mutation of interesting inputs such as: bit-flipping, byte-flipping, splicing, etc. It is a SAF technique with the following abstraction function:

$$\alpha_{afl}(\iota) \;=\; \{(bb_i^1, bb_j^1, log_2(n_1)), (bb_i^2, bb_j^2, log_2(n_2)), ..\}$$

where $(bb_i^*, bb_j^*, log_2(n_*))$ are pairs of basic blocks (bb) such that bb_j^* is visited right after bb_i^* for n_* number of times when the program processed the input ι and log_2 is the logarithm with base 2.

Dowser [11]: For input generation, Dowser uses constraint solving to generate interesting inputs.

It is also an SAF technique with a slightly different abstraction function than SAGE: instead of collecting all the constraints, it only collects the constraints at predetermined program points P_{Dowser}.

$$\alpha_{Dowser}(\iota) \;=\; \langle (c^1, b), (c^2, b), (c^3, b), ...\rangle \qquad (3)$$

where $c^* \in P_{Dowser}$ is a conditional statement of the program p reached by the input ι and b is the Boolean value (i.e., $b \in \{1, 0\}$), that indicates the result of the conditional statement.

Driller [19]: It uses both AFL method and constraint solving (similar to SAGE) for input generation. It is a SAF technique with the AFL's abstraction function i.e., α_{afl}.

VUzzer [17]: It generates inputs based on mutation and combinations of interesting inputs.

It is also a SAF technique with the following abstraction function, which is based on a scoring function ($score$) that is based on the basic blocks reached by the input.

$$\alpha_{VUzzer}(\iota) \;=\; \{(len(\iota), score(bb_w^1, bb_w^2, ...))\}$$

where $len(\iota)$ is the length of the input in bytes and bb_w^* is a pre-calculated weight of the basic block bb^* reached by the input ι.

Steelix [13]: This is a technique customized to fuzz magic byte based programs. For input generation, Steelix uses conditional mutation of the input bytes where a comparison failed. Steelix is actually a multi-abstraction fuzzer that combines AFL and the results of interesting comparison operations. Formally,

$$\begin{aligned}\alpha_{steel}(\iota) \;=\; &\{(bb_i^1, bb_j^1, lg_2(n_1)), (bb_i^2, bb_j^2, lg_2(n_2)), ...,\\ &(c^1, n_1), (c^2, n_2), (c^3, n_3), ...\}\end{aligned}$$

Here, the first part is similar to AFL, where $(bb_i^*, bb_j^*, log_2(n_*))$ are pairs of basic blocks (bb) such that bb_j^* is visited right after bb_i^* for n_* number of times.

The second part captures the results of interesting comparisons: where c^* is a conditional statement reached by the input and n_* is the number of bytes in the conditional statement that matched.

Angora [1]: Angora uses a single abstraction function similar to AFL's, with the addition of context sensitivity given by taking a hash of the callstack. Angora also generates new inputs using byte level taint tracking and a gradient decent algorithm for trying to satisfy conditional statements.

$$\begin{aligned}\alpha_{angora}(\iota) \;=\; &\{(bb_i^1, bb_j^1, h(stack_1), log_2(n_1)),\\ &(bb_i^2, bb_j^2, h(stack_2), log_2(n_2)), ..\}\end{aligned}$$

As shown, the provided formal model of fuzzing helps in understanding various fuzzing techniques in a systematic manner. Any fuzzing technique can be easily defined using our Fuzzing Parameters (Section III-D).

By describing existing fuzzing techniques using our formal model, one can see the following observations emerge:

Observation 1: Most current fuzzing techniques either develop new input generation techniques (e.g., Driller [19], DIFUZE [3]) or change, in tandem, both the abstraction function and the input generation technique (e.g., VUzzer [17], Angora [1], and Dowser [11]). In actuality, these two concepts are orthogonal.

Observation 2: Most existing fuzzing techniques are *Single-Abstraction Fuzzers* (SAF). That is, they use the same single abstraction function in every fuzzing iteration. However, there is no obvious reason why this *must* hold for all techniques.

This leads us to a natural research direction:

- With a fixed input generation technique, how does the chosen abstraction functions affect the effectiveness of fuzzing?
- Can a fuzzing strategy use *multiple* abstraction functions? Will it be more effective (given the same budget) than the corresponding SAF variation?

In the next section, we will describe a number of alternate abstraction functions that we will use to explore this direction of research.

IV. Abstraction Functions Explored

There are potentially infinite ways to abstract the concrete state space covered by an input on a program. Some aspects that abstraction functions could be based on are: (1) Code coverage, the set of basic blocks accessed [21], (2) Data access, the set of all global variables accessed, or (3) Function invocations, is the set of all library function called during program execution.

As mentioned in Section III-D, there could be several possible abstraction functions, where each could be effective in exploring a particular concrete state space of the program. Based on the requirements of effective exploration and performance overhead, we implemented six different abstraction functions, three of which are similar and have fine-grained granularity and the remaining three attempt to abstract different concrete state spaces of the program.

1) Basic Blocks *Abstraction (α_{bb}):* This is the most basic abstraction function which simply tracks the number of times each basic block was executed by the program. Instead of maintaining the raw counts, we use logarithmic counting set (as used in AFL[23]). Formally:

$$\alpha_{bb}(\iota) = \{(bb_i, log_2(c_i)), (bb_j, log_2(c_j), ...\}$$

Where, bb_* is the basic-block executed and c_* is the number of times corresponding basic block is executed when the program processed the input ι.

The intuition behind this abstraction is to capture the basic-block coverage achieved by an input on the program, under the assumption that more coverage yields more bugs.

2) Edges *Abstraction (α_{edge}):* This abstraction function increases the granularity of *Basic Blocks* abstraction by tracking the number of times (using logarithmic counting) an *edge* between basic blocks is executed. This is the same abstraction (Section III-D) used by the AFL fuzzer [23].

3) Block Triples *abstraction (α_{triple}):* The *Block Triples* abstraction function is similar to the *Edges* abstraction function, however instead of tracking edges (which is a pair of basic blocks), here we track all of the three consecutive basic blocks visited during the execution of the program.

The intuition here is that because the edges abstraction is successful (as shown by AFL), then perhaps increasing the granularity of the abstraction could increase its effectiveness.

4) Edge + Return Loc *Abstraction ($\alpha_{edgeret}$):* This abstraction, in addition to the *Edges* abstraction, also considers the calling function. In static-analysis terms, this is a 1-context sensitive version of the *Edges* abstraction. Formally,

$$\begin{aligned}\alpha_{edgeret}(\iota) &= \{(bb_i^1, bb_j^1, ret_1, log_2(n_1)),\\ &\quad (bb_i^2, bb_j^2, ret_2, log_2(n_2)), ..\}\end{aligned}$$

Where ret_* is the calling context under which the corresponding edge (bb_i^*, bb_j^*) was visited. The rest of the terms are the same as the *Edges* abstraction.

The intuition behind this abstraction is that if there is some function that performs a comparison, such as a `strcmp`, it is useful to satisfy that comparison when it is called in different contexts and not in a single context. For example, in Listing 1 a single function is used to check a three-byte header. A *Basic Blocks* or *Edges* abstraction can satisfy the check once because, as each successive byte is matched, the new input will be considered interesting. However, it will only match one header, because for a second header those blocks/edges will have already been seen in the utility function, and it would not be able to successively match the new bytes. On the other hand, the *Edge + Return Loc* abstraction will match the header multiple times, because it *includes the calling context.*

```
int check_header(char *data, char *header) {
  if (data[0] == header[0] && data[1] == header[1]
      && data[2] == header[2])
    return 1;
  else;
    return 0;
  }
}
int handle_data(char *data) {
  if (check_header(data, "TXT")) {
     // safe code
  }
  else if (check_header(data, "PDF")) {
     // BUG
  }
}
```

Listing 1: In this example, a utility function, `check_header` is used for multiple checks. An abstraction function that only tries to cover all edges will likely fail to pass both checks, because it will have already seen the edges within the utility function.

5) Function Context *Abstraction ($\alpha_{context}$):* In this abstraction function, we attempt to capture the context of the executed functions. As the context could be potentially unlimited, we limit the length of the context to four. We capture the context as the sequence of the last four return address on the call-stack at the entry of each executed function. In the case where the call-stack has less than four return addresses we use *null* instead. This is done by xoring the callstack entries.

The motivation for this abstraction is that, in many real-world vulnerabilities, the context of certain function invocations is critical. For example, in a JavaScript engine, the JavaScript code can add or remove elements in an array. However, these operations might not be safe if the code is called from inside a `sort` function which does not handle a changing array size (as in CVE-2013-0997 [5]).

6) Method Calls *Abstraction(α_{omcp}):* This abstraction function is specialized for object-oriented programs, and we capture the pairs of methods executed on the same object instantiation. Consider an example, where we have an object `foo` with methods `A()`, `B()`, and `C()`. If the execution of a program given input ι results in the following method invocation sequence: `foo.A()`, `foo.B()`, `foo.C()`, then we will add the pairs `A-B`, `A-C`, `B-C`, to the counts. If there is a second object `bar` that is the same class as `foo` and the program execution is: `foo.A()`, `bar.B()`, `foo.C()`, `bar.B()` we will add the pairs `A-C`, `B-B` to the counts because the method calls are tracked on a per-object basis.

7) Steelix *($\alpha_{steelix}$):* Although this is actually a multi-abstraction as explained in Section III-D, it is included here as it is one of the strategies we explore in our evaluation. The details are previously explained, but briefly, *Steelix* uses two abstraction functions, edges, and one which looks at comparisons with important values.

A. Multi-Abstraction

As demonstrated in Section III-D, most existing fuzzing techniques are Single Abstraction Fuzzing (SAF). In this paper, we explore combining multiple abstraction functions so that they can provide new inputs to each other and combine the strengths that each abstraction function provides. For example, let us consider how the *Edges* and *Method Calls* abstractions described previously might pair well together. The *Method Calls* abstraction instruments method calls on objects and will consider different arrangements of method calls interesting, however it might not be able to trigger a particular method call in the first place. The *Edges* abstraction might easily find that new method call, however it may not find the different arrangements of the method calls interesting.

To combine abstractions, we chose to run them as parallel fuzzers and share inputs between them, similar to the ensemble fuzzing strategy presented by Chen et. al [2] and Wang et. al [22]. In other words, each fuzzer will use its own abstraction function but consider all interesting inputs that all fuzzers discover. Of course there are other ways of using multiple abstraction functions, such as switching between them, although exploring such other ways of combining them is out of scope of this paper.

V. Implementation

We implemented each of the abstraction functions including the Mult-Strategy abstraction using American Fuzzy Lop, as it is one of the most popular and effective open source fuzzers. To implement the Multi-Strategy abstractions we run each input selection strategy in parallel with separate (distinct) fuzzers, however the found interesting inputs are shared between each fuzzing instance.

VI. Evaluation

Lacking the formalism contributed by this paper, existing approaches in fuzzing do not maintain a separation between the abstraction function utilized and other details of the respective approaches. Additionally, it seems that current fuzzing techniques do not tend to leverage *multiple* abstraction functions. In this section, we explore both of these oversights, evaluating the effectiveness of alternate abstraction functions as well as the combination of multiple abstraction functions. We attempt to answer the following research questions.

RQ1: Does the choice of abstraction function affect the bug finding capabilities?

RQ2: How effective is it to combine different abstraction functions?

When evaluating these research questions, it is important to keep in mind we are not only looking to see if the choice of abstraction functions or combination thereof results in more crashes, but also to see if the bugs that are found differ. That is, if one abstraction function finds less bugs, but finds bugs not found by another than it is still interesting and would be worth applying.

Also, it is important to reiterate, as stated in Observation 2 in Section III, that the choice of abstraction function (or functions) is *orthogonal* to other aspects of the fuzzing process. That is, though symbolically-assisted approaches (such as Driller [19]) would have a scaling effect on the numbers reported in this section, they would not effect the *relations* between these numbers. As such, we evaluate our system using a modification of the American Fuzzy Lop fuzzer.

TABLE I
Multistrategy Fuzzer Configurations.

Selection Strategy	Abstraction Functions
Multi-Strategy1	*Basic Blocks*, *Edges*, *Block Triples*, *Edge + Return Loc*, *Function Context*, *Method Calls*
Multi-Strategy2	*Steelix*, *Basic Blocks*, *Edge + Return Loc*, *Function Context*

A. Dataset

To explore our research questions, we chose a varied dataset that is amenable to large-scale experiments. Specifically, we use the dataset of vulnerable programs produced for the DARPA Cyber Grand Challenge (CGC) [6] for our experimentation. These binaries contain a very diverse set of functionalities and variable complexity, and guarantee the presence of known bugs [15], [14], providing a perfect testing ground for our system. After filtering out challenges that involved multiple binaries (which AFL cannot currently fuzz) and challenges which failed to build for Linux (using a port of the CGC dataset to that platform [16]), there were 217 different binaries that we used.

To evaluate the impact of abstraction functions on *real-world* software, we evaluated them on Objdump and ImageMagick in Section VI-E.

B. Experimental Setup

We evaluated each binary in the CGC dataset using each of the seven selection strategies detailed in Section IV). Each of these single-strategy fuzzing *configurations* was run for 8 hours on 12 cores, giving each configuration 96 CPU hours of experimentation time. This is well over the minimum time suggested by [12].

We also evaluated two *multi-abstraction fuzzing* (MAF) configurations that combines multiple select abstraction functions. In each of these, the abstractions were run in parallel (with cross-fuzzer synchronization of $\widetilde{I}$) for 8 hours, with 12 cores total divided equally between the fuzzers. Note that this is the same amount of resources provided for testing the single-abstraction fuzzers. The abstractions used in the MAF configurations are shown in Table I. We chose one configuration (*Multi-Strategy1*) which included every abstraction function except *Steelix*, and one MAF configuration with *Steelix* (*Multi-Strategy2*). The abstraction functions chosen for *Multi-Strategy2* aimed to pick the a smaller, but varied set of selection strategies.

C. Crash Numbers

Table III shows the total number of binaries crashed by each SAF configuration, along with the MAF configurations. Confirming our intuition, both multi-strategy fuzzers performed better than any of the single-strategy fuzzers that were part of them. The three best fuzzers (*Multi-Strategy2*, *Steelix*, *Multi-Strategy1*) were all MAF's, indicating that MAF fuzzers perform better than their SAF counterparts. The *Steelix* strategy performed very well, crashing a total of 127 binaries. This

TABLE II
THIS TABLE SHOWS THE PERCENTAGE OF BINARIES THAT ARE CRASHED BY THE CONFIGURATION IN THE CORRESPONDING ROW THAT ARE *also* CRASHED WHEN USING THE CONFIGURATION IN THE CORRESPONDING COLUMN. THE VALUE OF EACH CELL AT ROW m AND COLUMN n IS: $\frac{C_{AF_m} \cap C_{AF_n}}{C_{AF_m}} * 100$, WHERE C_k IS THE TOTAL BINARIES CRASHED WHEN USING THE ABSTRACTION FUNCTION k, AF_m, AND AF_n ARE THE ABSTRACTION FUNCTIONS OF ROW m AND COLUMN n RESPECTIVELY. THE CELLS IN THE TABLE ARE SHADED BASED ON THE VALUE VERYHIGH 100-95, HIGH 95–90, MEDIUM 90–80, AND LOW < 80.

	Basic Blocks	*Edges*	*Block Triples*	*Edge + Return Loc*	*Function Context*	*Method Calls*	*Multi-Strategy1*	*Steelix*	*Multi-Strategy2*
Basic Blocks	—	94.51	94.51	96.70	80.22	61.54	96.70	94.51	95.60
Edges	94.51	—	94.51	96.70	81.32	63.74	96.70	97.80	96.70
Block Triples	93.48	93.48	—	97.83	80.43	63.04	98.91	97.83	97.83
Edge+Ret Loc	87.13	87.13	89.11	—	74.26	58.42	99.01	94.06	98.02
Function Context	93.59	94.87	94.87	96.15	—	73.08	96.15	100.00	100.00
Method Calls	91.80	95.08	95.08	96.72	93.44	—	98.36	98.36	98.36
Multi-Strategy1	83.02	83.02	85.85	94.34	70.75	56.60	—	90.57	94.34
Steelix	67.72	70.08	70.87	74.80	61.42	47.24	75.59	—	97.64
Multi-Strategy2	65.91	66.67	68.18	75.00	59.09	45.45	75.76	93.94	—

TABLE III
RESULTS OF EACH OF FUZZING USING THE DIFFERENT ABSTRACTION FUNCTIONS ON THE CGC DATASET.

Selection Strategy	Type	Number Crashes	Mean Block Coverage	Mean Execs per sec
Basic Blocks	SAF	91	38.1%	625
Edges	SAF	91	38.5%	620
Block Triples	SAF	92	39.0%	412
Edge + Return Loc	SAF	101	39.5%	555
Function Context	SAF	78	35.5%	569
Method Calls	SAF	61	28.6%	469
Multi-Strategy1	MAF	106	39.5%	551
Steelix	MAF	127	47.2%	581
Multi-Strategy2	MAF	132	48.1%	530

result was expected because it is designed to recover strings and magic numbers, of which there are many in the CGC. The *Multi-Strategy2* strategy performed the best, crashing 132 of the 217 binaries. The one SAF that performed the best was *Edge + Return Loc*, crashing 102 binaries, 10 more than any other SAF. This result shows that adding some callstack context to the abstraction seems to improve it's bug-finding capabilities.

The *Method Calls* configuration was the least effective with only 55 crashes. As explained in Section IV, this abstraction function is specialized for object-oriented programs, while most of the CGC binaries are not object-oriented.

These results allows us to infer an answer to Research Question 1 (RQ1).

Answer for RQ1: The choice of an abstraction function is important in fuzzing, and an inappropriate abstraction function (e.g., *Method Calls* on non-object-oriented code) can seriously impair fuzzing effectiveness.

The relatively poor performance of the *Method Calls* and *Function Context* SAF configurations does not mean that these abstractions are useless. In fact, we found in our evaluation, when combined with other abstraction functions in *Multi-Strategy1*, it enabled the detection of a crash in Modern_Family_Tree which no other configuration found.

Table II shows a fine-grained comparison of different configurations. It contains the percentage of binaries that are crashed by the configuration in the corresponding row that are *also* crashed when using the configuration in the corresponding column. Specifically, the value of each cell at row m and column n is computed as: $\frac{C_{AF_m} \cap C_{AF_n}}{C_{AF_m}} * 100$, where C_k is the total binaries crashed when using the abstraction function k, AF_m, and AF_n are the abstraction functions of row m and column n respectively. This table allows the comparison of the different abstraction functions against each other. For a given pair of abstraction functions α_m and α_n), we can make following observations based on the value of a cell (m, n), which is the value at row m and column n:

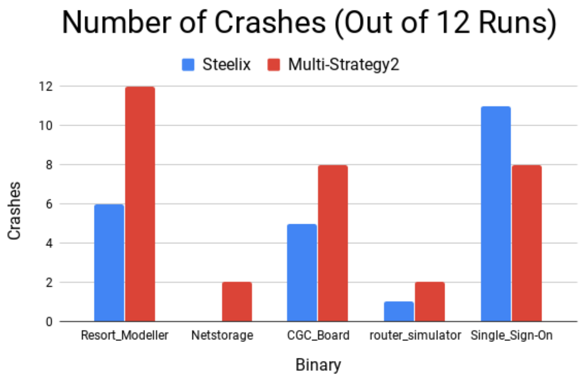
- A higher value at the cell (α_m, α_n) indicates that α_n encompasses the effectiveness of α_m.
- A higher value at the cell (α_n, α_m) indicates that α_m encompasses the effectiveness of α_n.
- A higher value at both the cells (α_m, α_n) and (α_n, α_m) indicates that the abstraction functions α_m and α_n have similar bug finding abilities.
- A lower value at both the cells (α_m, α_n) and (α_n, α_m) indicate that the abstraction functions find different bug types and these are the good candidates to be combined.

The *Multi-Strategy1* configuration was able to leverage the capabilities of the six abstraction functions that it utilized, and this can be seen from the high percentages in the cells of the *Multi-Strategy1* column and relatively low percentages in the cells of the *Multi-Strategy1* row. This is true despite the fact that the individual abstraction functions in *Multi-Strategy1* receive only a fraction of the time they had individually.

If we look at the *Steelix* column we see that the only SAF's that were not very highly encompassed by it were *Edge + Return Loc* and *Basic Blocks*. *Basic Blocks* being a more coarse-grained abstraction might've done better in cases where *Steelix* produced too many paths to process. This result also implies that combining *Steelix*, *Basic Blocks*, *Edge + Return Loc*, *Function Context* in the *Multi-Strategy2* configuration is a good choice.

Looking specifically at the column for *Multi-Strategy2*, we see that it encompassed all the other strategies fairly well; *Multi-Strategy1* was the only strategy that it didn't find at least 95% of the same crashes. This implies that although we were able to capture most of the bug finding capabilities of the other

Fig. 1. This graph shows the results of re-running the *Steelix* and *Multi-Strategy2* fuzzer configurations 12 times on five select binaries that were only found by *Multi-Strategy2*. These results do not include the original large-scale test.



strategies with this combination, we still missed some bugs that a different combination of strategies got.

Answer for RQ2: The results show that both *Multi-Strategy1* and *Multi-Strategy2* configurations were more effective at finding crashes than their individual abstraction functions. Therefore, combining abstraction functions, is an effective technique to enhance fuzzing.

Abstraction functions based on basic blocks (*Basic Blocks*, *Edges*, *Block Triples*, and *Edge + Return Loc*) all show similar bug finding abilities, and this is evident from the high percentage in the cells of the corresponding abstractions. Finally, it is interesting to see that having more fine-grained abstractions (e.g. block triples, vs edges) does not necessarily improve the effectiveness.

D. Different Strategies Repeatedly Crash Different Binaries

In the evaluation, we saw plenty of cases in which a binary is crashed by one configuration and not another. This is seen in Table II, anywhere where the overlap is not 100%. We want to evaluate whether these results are random or if binaries exist where one configuration is for sure better at finding that crash.

First we explore *Multi-Strategy2* vs *Steelix*, to know if the binaries crashed by the first and not the second are random or repeatable. To test this, we picked five binaries that were found by *Multi-Strategy2* and not *Steelix* and explored what happened if we repeated the test 12 additional times. These results are shown in Figure 1. Note that the original fuzzing run is not included to make sure the results aren't biased. We see that of the five binaries that in only one case did *Steelix* do better in our repeated evaluation. This result shows that *Multi-Strategy2* outperforming it on these binaries is repeatable.

Even when comparing individual abstraction functions we were frequently able to find examples where one abstraction function repeatedly did better than others. Due to cost and time constraints we weren't able to re-run all tests twelve times like above, but here's some examples we tested.

TABLE IV
RESULTS OF FUZZING USING DIFFERENT ABSTRACTION FUNCTIONS ON IMAGEMAGICK AND OBJDUMP. NOTE THAT THERE WERE NO CRASHES ON IMAGEMAGICK WITHOUT SEEDS SO THOSE COLUMNS ARE OMITTED. RESULTS SHOWN ARE THE MEDIAN OF SIX RUNS.

	ImageMagick 7.0.4-2 (seeds)	Objdump 2.26.1 (seeds)	Objdump 2.26.1 (no seeds)
Selection Strategy	Median crashes	Median crashes	Median crashes
Basic Blocks	0.5	**7.0**	3.0
Edges	0.0	**7.0**	4.0
Block Triples	1.0	5.5	**4.5**
Edge + Return Loc	1.5	6.5	4.0
Function Context	1.0	4.0	1.5
Method Calls	0.0	3.0	1.0
Multi-Strategy1	1.0	5.0	2.5
Steelix	1.0	6.0	3.5
Multi-Strategy2	**2.0**	6.0	**4.5**

- Dive_Logger was crashed at least 10/12 times by *Edge + Return Loc* and each of the MAF's that include *Edge + Return Loc*, but rarely by other strategies.
- Modern_Family_Tree was crashed 11/12 times by *Multi-Strategy1*, but no more than 3/12 times by any other strategy (including *Multi-Strategy2*).
- Neural_House was rarely crashed by *Edges* and *Basic Blocks*, but frequently by *Block Triples*, *Edge + Return Loc* and *Steelix*.

These results that sometimes a fuzzer which in general doesn't find the most crashes might be better at a specific binary. It also indicates that the incomplete overlap we saw in Table II is a product of the different performance of the fuzzing configurations.

E. Real World Programs

We evaluated the effectiveness of the individual abstraction functions, as well as the effectiveness of the multi-abstraction fuzzers on two real-world programs: Objdump-2.26.1, and ImageMagick-7.0.4-2. These are programs that have been used in other fuzzing evaluations [12].

We ran each fuzzer configuration described previously six times on each of these programs. Each run was using 12 cores for 8 hours for a total of 96 core-hours. Then we analyze the results in terms of unique number of crashes, where unique is defined in terms of the crashing callstack. Note that these do not represent unique bugs, as attributing crashes to bugs is out of scope of this paper. These programs were fuzzed both with and without seeds.

The results are shown in Table IV. As we can see from the table, the strategy that does the best depends both on the binary being fuzzed and whether or not seeds are provided. *Basic Blocks* and *Edges* do better than any other configuration on Objdump with seeds, however, they both are significantly less effective without seeds and perform very poorly on ImageMagick. Looking into the results, it seems that for Objdump, especially with seeds, these two strategies had generated significantly less inputs compared to say *Steelix*, and executed inputs about 20% faster. Since the extra capabilities of *Steelix* and *Edge + Return Loc* weren't necessary to trigger additional coverage (especially when given seeds!) the faster basic sensitivities were the best.

One consistent trend across all of them is that *Multi-Strategy2* does very well, even if it is not always the best. This confirms our earlier results that it is widely applicable on a range of binaries. *Multi-Strategy2* has the abilities of *Basic Blocks* to explore lots of inputs quickly and that of *Steelix* and *Edge + Return Loc* to deeply explore. Another interesting point is that *Edge + Return Loc* did well on all four tests. This implies that some callstack context is important for fuzzers to be able to explore lots of the state-space.

VII. Discussion

We saw in our evaluation of fuzzing strategies that combining different abstraction functions allows them to complement each other by sharing inputs between them. This resulted in the multi-abstraction fuzzers crashing more binaries than any single strategy from their components. Even though the *Multi-Strategy2* configuration found the most crashes in our tests, there were programs in which another configuration was more likely to crash. Future work could be to try and automatically determine the best fuzzer configuration to use. Also, as mentioned in the framing of the evaluation, other input generation techniques, such as Driller, could also be used on top of this to improve results for all abstractions in our evaluation.

Despite finding that *Multi-Strategy2* found the most crashes in our tests, we also saw programs in which a different configuration was more likely to find a crash, as was discussed in Section VI-D. One takeaway of this is that although a particular fuzzer might find less crashes than another it still might be useful if it finds *different* crashes. Consider *Edge + Return Loc*, alone it finds less crashes in number than *Steelix* but it also found crashes in *different* programs.

VIII. Conclusion

In this paper, we have presented the first formalization of input evaluation in fuzzing, borrowing concepts from the field of static analysis. This formalization can immediately be used as an effective base for future research: by reasoning about which concepts of the formalization have been explored by current work (and the more salient question of which have not been), we identified that the impact of different *abstraction functions* on fuzzing outcomes is unexplored in current work.

Thus, we performed an investigation into alternate abstraction functions for fuzzers. We identified seven input abstraction functions with various levels of granularity and evaluated them on a large dataset from the DARPA Cyber Grand Challenge and on two real-world programs. The results show that the choice of an abstraction function is important and can affect the effectiveness of fuzzing. Furthermore, we show that combining different abstraction functions is superior to using just one.

Keeping with the scientific spirit, we open-source the resulting abstraction-modular fuzzer.

References

[1] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[2] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1967–1983.

[3] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2123–2138.

[4] 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*. ACM, 1977, pp. 238–252.

[5] CVE-2013-0997, https://packetstormsecurity.com/files/123229/Apple-Security-Advisory-2013-09-12-2.html, 2013.

[6] DARPA, "Darpa cyber grand challenge," 2016, http://archive.darpa.mil/cybergrandchallenge/.

[7] J. Drake, "Stagefright: Scary code in the heart of android," *BlackHat USA*, 2015.

[8] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 474–484. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070546

[9] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 206–215. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375607

[10] Google, "syzkaller - linux syscall fuzzer," 2017, https://github.com/google/syzkaller.

[11] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowser: a guided fuzzer to find buffer overflow vulnerabilities," in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 49–64.

[12] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.

[13] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 627–637. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106295

[14] LungeTech, "Cgc data archive for finals," 2017, http://www.lungetech.com/cgc-corpus/cwe/cfe/.

[15] ——, "Cgc data archive for qualifiers," 2017, http://www.lungetech.com/cgc-corpus/cwe/cqe/.

[16] T. of Bits, "Darpa challenge binaries on linux, os x, and windows," 2017, https://github.com/trailofbits/cb-multios.

[17] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[18] S. Rawat and L. Mounier, "Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 531–533.

[19] 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 of the 2016 Network and Distributed System Security Symposium*, 2016.

[20] G. Vranken, "libfuzzer-gv: new techniques for dramatically faster fuzzing," https://guidovranken.wordpress.com/2017/07/08/libfuzzer-gv-new-techniques-for-dramatically-faster-fuzzing/, 2017.

[21] D. Vyukov, "Kcov: Kernel coverage," 2017, https://lwn.net/Articles/671640/.

[22] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2019, pp. 1–15.

[23] M. Zalewski., "American fuzzy lop," 2017, http://lcamtuf.coredump.cx/afl/technical_details.txt.