REDQUEEN: Fuzzing with Input-to-State Correspondence

Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik and Thorsten Holz
Ruhr-Universität Bochum

***Abstract*—Automated software testing based on fuzzing has experienced a revival in recent years. Especially feedback-driven fuzzing has become well-known for its ability to efficiently perform randomized testing with limited input corpora. Despite a lot of progress, two common problems are magic numbers and (nested) checksums. Computationally expensive methods such as taint tracking and symbolic execution are typically used to overcome such roadblocks. Unfortunately, such methods often require access to source code, a rather precise description of the environment (e.g., behavior of library calls or the underlying OS), or the exact semantics of the platform's instruction set.**

In this paper, we introduce a lightweight, yet very effective alternative to taint tracking and symbolic execution to facilitate and optimize state-of-the-art feedback fuzzing that easily scales to large binary applications and unknown environments. We observe that during the execution of a given program, parts of the input often end up directly (i.e., nearly unmodified) in the program state. This *input-to-state correspondence* can be exploited to create a robust method to overcome common fuzzing roadblocks in a highly effective and efficient manner. Our prototype implementation, called REDQUEEN, is able to solve magic bytes and (nested) checksum tests automatically for a given binary executable. Additionally, we show that our techniques outperform various state-of-the-art tools on a wide variety of targets across different privilege levels (kernel-space and userland) with no platform-specific code. REDQUEEN is the first method to find more than 100% of the bugs planted in LAVA-M across all targets. Furthermore, we were able to discover 65 new bugs and obtained 16 CVEs in multiple programs and OS kernel drivers. Finally, our evaluation demonstrates that REDQUEEN is fast, widely applicable and outperforms concurrent approaches by up to three orders of magnitude.

I. INTRODUCTION

Fuzzing has become a critical component in testing the quality of software systems. In the past few years, smarter fuzzing tools have gained significant traction in academic research as well as in industry. Most notably, american fuzzy lop (AFL [44]) has had a significant impact on the security landscape. Due to its ease of use, it is now convenient to more thoroughly test software, which many researchers and developers did. On the academic side, DARPA's Cyber Grand Challenge (CGC) convincingly demonstrated that fuzzing remains highly relevant for the state-of-the-art in bug finding: all teams used this technique to uncover new vulnerabilities.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23xxx
www.ndss-symposium.org

Following CGC, many new fuzzing methods were presented which introduce novel ideas to find vulnerabilities in an efficient and scalable way (e.g., [10], [16], [19], [31], [34]–[38]).

To ensure the adoption of fuzzing methods in practice, fuzzing should work with a minimum of prior knowledge. Unfortunately, this clashes with two assumptions commonly made for efficiency: (i) the need to start with a good corpus of seed inputs or (ii) to have a generator for the input format. In absence of either element, fuzzers need the ability to *learn* what interesting inputs look like. Feedback-driven fuzzing, a concept popularized by AFL, is able to do so: Interesting inputs which trigger new behavior are saved to produce more testcases, everything else is discarded.

A. Common Fuzzing Roadblocks

To motivate our approach, we first revisit the problem of efficiently uncovering new code, with a focus on overcoming common fuzzing roadblocks. In practice, two common problems in fuzzing are magic numbers and checksum tests. An example for such code can be seen in Listing 1. The first bug can only be found if the first 8 bytes of the input are a specific magic header. To reach the second bug, the input has to contain the string "RQ" and two correct checksums. The probability of randomly creating an input that satisfies these conditions is negligible. Therefore, feedback-driven fuzzers do not produce new coverage and the fuzzing process stalls.

```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);

/* nested checksum example */
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]=='R' && input[17]=='Q')
      bug(2);
```

Listing 1: Roadblocks for feedback-driven fuzzing.

In the past, much attention was paid to address such roadblocks. Different approaches were proposed which typically make use of advanced program analysis techniques, such as taint tracking and symbolic execution [12], [13], [16], [22], [23], [26], [35], [38], [40]. Notably, both ANGORA [16] and T-FUZZ [34] fall into this category. These approaches usually require a rather precise description of the environment (e.g., behavior of library calls or the underlying OS) and the exact semantics of the platform's instruction set. As a result, it is hard to use this methods on targets that use complex instruction set extensions (i. e., floating point instructions) or uncommon

libraries and operating systems. Therefore, such approaches are the polar opposite of the approach pioneered by AFL: to a large extend, AFL's success is based on the fact that it makes few assumptions about the program's behavior. Based on this insight, we investigate a novel fuzzing method that excels at increasing code coverage on arbitrary targets, ranging from open-source userland programs to closed-source OS kernels. We demonstrate that this approach can outperform existing fuzzing strategies.

B. Our Approach: Input-to-State Correspondence

In this paper, we propose a novel and lightweight method that—in many cases— is able to replace the two complex analysis primitives taint tracking and symbolic execution. In contrast to the two aforementioned techniques, our method is easy to implement and scales to large, complex targets and diverse environments. Our approach is based on a simple yet intuitive observation: in many cases, parts of the input directly correspond to the memory or registers at run time. Hence, there is a strong *input-to-state correspondence* between the input and the current program state. This can be exploited to implement an efficient fuzzing approach. In practice, most programs apply only a small number of decoding steps to the input, before the inputs data is used. We found that the set of encoding schemes used by real-world programs is typically rather small: normally, these values are directly used in the context of hard condition such as checks for a specific header value (magic bytes). For example, it is common that input bytes are interpreted as little-endian integers and then directly compared against a checksums or specific magic bytes.

We exploit this observation by tracing the program to observe values used in compare instructions. By "colorizing" the input with random bytes, we create a very lightweight approximation to taint tracking. Then, we speculate that we are able to control these values by changing the corresponding input bytes. Finally, we use the fast fuzzing process to verify whether we triggered new and potential interesting behavior. Similarly, we swiftly discard false positives which arise from this over-approximation. This method allows us to skip complex sections of the code such as API calls or unknown instructions, which would otherwise be difficult to handle for taint tracking or symbolic execution. As a result, even inputs which pass through unknown library functions, large data-dependent loops, and floating point instructions do not significantly reduce the quality of the results. We continue to use the same principle to implement a patching-based solution to handle checksum tests. In contrast to similar approaches, our approach entirely avoids symbolic execution, while always maintaining a queue of inputs with valid checksums and no false positives.

Feedback-driven fuzzers typically use the same instrumentation across all executions to measure code coverage and other feedback information. For example, VUZZER uses its full taint tracking capabilities on each input generated. Since most fuzzing executions (millions to billions) happen on top of the same few inputs (thousands to hundreds of thousands), we propose to incorporate analysis into feedback-driven fuzzing differently: we separate the more expensive analysis process from the fuzzing process. In our case, we perform the expensive search for path-specific input-to-state correspondences only once per new input found. All actual fuzzing is then performed without this additional overhead. We found that this approach greatly reduces the cost associated with more expensive analysis and allows the remaining fuzzing process to take advantage of the knowledge gained during the analysis phase.

While our approach can be seen as an approximation to taint tracking and symbolic execution, our evaluation results are quite competitive with tools using more expensive "real" taint tracking and symbolic execution. To perform our evaluation, we implemented a prototype of our approach, called REDQUEEN, that can handle binary-only targets. Our empirical evaluation on the GNU `binutils` suite demonstrates, that our approach is, *in all cases*, able to cover significantly more code than existing tools. Measuring the time to equal coverage yields speedups in the range of *5x to 5000x* when compared to VUZZER [35] and KLEE [12], as well as *2x to 200x* against AFLFAST [10] and LAF-INTEL [2]. In addition, we are the first to find significantly more bugs (2600) than listed in the LAVA-M data set (2265). In total, we found an additional 335 unlisted bugs, yielding over 114% of all listed bugs. We only missed two of the 2265 listed vulnerabilities in the LAVA-M data set ($>99.9\%$ bug coverage).

Moreover, our technique avoids a significant amount of implementation and performance overhead typically associated with taint tracking and symbolic execution. Therefore, our approach is applicable to a much more diverse set of targets than aforementioned techniques, which require a detailed environment model. In fact, REDQUEEN is able to fuzz programs with no need for source code or platform knowledge, as we demonstrate by applying REDQUEEN to both kernel- as well as user-space targets. In our evaluation, REDQUEEN found 10 bugs in 2 different Linux file system drivers and 55 bugs in 16 user-space programs and software libraries. Additionally, 16 CVEs have been assigned for some of the more critical issues we uncovered.

C. Contributions

In summary, we make the following contributions:

- We introduce the concept of input-to-state correspondence as a novel principle to significantly accelerate feedback-driven fuzzing.
- We show that input-to-state correspondence can be used instead of taint tracking and symbolic execution to solve hard fuzzing problems such as dealing with magic numbers, dynamic multi-byte compares, and (nested) checksums without introducing any false positives. The resulting mutation operator is more efficient than all other mutation operator used by AFL (measured in new paths found over time used).
- We built a prototype implementation of our method in a tool called REDQUEEN. Our comprehensive evaluation results demonstrate that REDQUEEN outperforms all state-of-the-art fuzzing tools in several metrics.

To foster research on this topic, we release our fuzzer at https://github.com/RUB-SysSec/redqueen.

II. Related Work

Fuzzing has been an active field of research for decades. Initially, much attention was focused on improving black-box fuzzing (e.g., fuzzing strategies in which the fuzzer does not inspect the program internals and treats it as a black-box). Which results in improved scheduling algorithms [14], [36], [41] and more effective mutation or input generation strategies [4], [29]. Even machine learning techniques were investigated to infer semi-valid inputs as test cases [8], [24], [27]. Recently, more focus was put on white- and gray-box fuzzing. Usually, the program under test is instrumented to generate some kind of feedback (such as code coverage). The archetypal gray-box fuzzer is AFL [44]. It uses coverage as a feedback mechanism to learn which inputs are interesting and which do not trigger new behavior. Much recent work is based on AFL. The scheduling of AFL was analyzed and improved by AFLFast [10]. CollAFL [19] and InsTrim [30] improve the performance of AFL, by decreasing the probability that two distinct paths are considered the same. The performance of the fuzzers itself were improved in many different ways [25], [42]. One notable example is go-fuzz [39] which was developed independently and uses a similar idea as Redqueen. The OSS-Fuzz project scaled the AFL fuzzing model to large computing clusters and discovered a significant amount of vulnerabilities in highly relevant open-source software [1]. HonggFuzz [6] and kAFL [37] use algorithms inspired by AFL and modern CPU extensions to demonstrate how binary-only targets can be fuzzed in an efficient manner. In this work, we aim at solving problems commonly addressed by white-box fuzzing techniques. We will, therefore, use this section to differentiate our approach from existing work on fuzzing.

A. Symbolic/Concolic Execution-based Fuzzing

Several gray- or white-box fuzzing tools make use of symbolic execution to improve test coverage [12], [13]. Symbolic execution can find vulnerabilities that are very hard to trigger randomly, or even, with clever heuristics. However, it also tends to become very slow on large targets and state explosion has to be carefully taken into account. One common approach to handle state explosion is to use concolic execution [21]–[23], [26], [33], [38], [40]. In concolic execution, the program path is constrained to a concrete path while the solver either tries to trigger bugs on this path or to uncover a new path. This approach greatly reduces the number of states that are being explored and can—at least in some cases—be used to reduce the complexity of the formulas encountered by replacing complex expressions with their concrete values. Additionally, symbolic execution has often been motivated by the need to solve magic byte checks. Our results show that, in our empirical evaluation, a much simpler approach is often sufficient to solve these cases.

B. Taint-based Fuzzing

Similar to symbolic execution, taint tracking is commonly used by fuzzing tools. It allows learning which parts of the input are affecting certain operations. In the past, taint tracking was used to identify and focus on parts of the input that are used as magic bytes [35], addresses [20], [26], [35], or integers that might overflow [33]. In this paper, we show that input-to-state correspondence can often be used as an approximation to taint tracking and does solve these common problems much more efficiently. Recently, another taint-based fuzzing approach called Angora [16] was proposed. Similarly to our approach, Angora uses the expensive taint tracking step only sparsely to overcome hard-to-solve conditions. In contrast, Angora relies on source code access and a special compiler pass to perform efficient taint tracking, while we propose a binary-level fuzzer. Moreover, Angora cannot handle checksums.

C. Patching-based Fuzzing

Most symbolic execution based tools are able to generate valid checksums, but cannot use the faster fuzzing component to explore the code after the check. Some fuzzers try to patch hard checks to make a given program easier to fuzz. Three examples of fuzzing methods that follow this approach are flayer [18], TaintScope [40], and T-Fuzz [34]. Since we also patch checksum tests to be able to fuzz efficiently, a more thorough discussion of these tools is provided in Section III-B. Our idea for checksum handling is inspired by both flayer and TaintScope. However, flayer needs an explicit list of conditional branches to patch. In addition, the user has to fix the input after the fuzzing process. In contrast, TaintScope is able to infer the list of checks automatically and patch all hard-to-solve branches during fuzzing. Then—after finishing the fuzzing process—TaintScope uses symbolic execution to fix crashing inputs. Similar to TaintScope, our process is entirely automated. In contrast, however, we use our idea of input-to-state correspondence to avoid the complex and often brittle taint tracking and symbolic execution methods. T-Fuzz also uses an approach related to TaintScope: the program is transformed to reach code after hard-to-solve conditions; broken checks are later fixed using symbolic execution. The improvement in performance that Redqueen provides over T-Fuzz can be explained by the fact that T-Fuzz needs to spawn new fuzzing instances for each hard-to-reach part of the code. Additionally, T-Fuzz does not remove false positives during fuzzing. Hence, the number of fuzzing instances that are working on dead ends can grow nearly unbounded. In contrast, our approach avoids these scalability issues all together by always maintaining a queue of valid inputs. Therefore Redqueen neither produces, nor spends time on false positives.

D. Binary-Only Fuzzers

Many fuzzers such as AFL, laf-intel and Angora need source code access to add the necessary instrumentation and patches. As a result, proprietary systems cannot be analyzed with these fuzzers. To overcome this limitation, some fuzzers use other mechanisms to obtain feedback. AFL has spawned multiple forks that use `PIN` [32], `DynamoRIO` [11] or `QEMU` [9] to obtain coverage information. Similarly, fuzzers like VUzzer, TaintScope, flayer, T-Fuzz and Driller make use of various dynamic binary instrumentation tools. We found that the fastest binary-only fuzzer is AFL with `QEMU`, which is significantly slower (in executions per second) than AFL with compile-time instrumentations.

E. The AFL Family

Due to the overwhelming success of the AFL design, many different tools are heavily based on AFL [1], [2], [5],

[10], [16], [31], [37], [38]. Our work is based on KAFL—an AFL-like fuzzer—and, therefore, it is paramount to have a rough understanding of the design of AFL. Generally speaking, fuzzers from the AFL family have three important components: (i) the queue, (ii) the bitmap, and (iii) the mutators. The queue is where all inputs are stored. During the fuzzing process, an input is picked from the queue, fuzzed for a while, and, eventually, returned to the queue. After picking one input, the mutators perform a series of mutations. After each step, the mutated input is executed. The target is instrumented such that the coverage produced by the input is written into a bitmap. If the input triggered new coverage (and, therefore, a new bit is set in the bitmap), the input is appended to the queue. Otherwise, the mutated input is discarded. The mutators are organized in different stages. The first stages are called the *deterministic stages*. These stages are applied once, no matter how often the input is picked from the queue. They consist of a variety of simple mutations such as "try flipping each bit". When the deterministic stages are finished or an input is picked for the second time, the so called *havoc phase* is executed. During this phase, multiple random mutations are applied at the same time at random locations. Similarly, if the user provided a dictionary with interesting strings, they are added in random positions. Linked to the havoc stage is the *splicing stage*, in which two different inputs are combined at a random position.

III. Input-to-State Correspondence

In this section, we introduce a novel fuzzing method based on the insight that programs have a strong input-to-state correspondence. We observe that—for a very large number of programs—values from the input are directly used at various states during the execution. By observing these values, we can perform educated guesses as to which offsets to replace (resembling a very lightweight taint tracking) and which value to use (similar to symbolic execution based approaches). We can exploit this relation to deal with challenging fuzzing problems such as magic bytes and (even nested) checksums. We explain the different building blocks of our method and discuss how they address the challenging fuzzing problems we introduced earlier.

A. Magic Bytes

The first roadblock we tackle are magic bytes. A typical example for this class of fuzzing problems is shown in Listing 2; it represents an excerpt of our running example introduced in Listing 1. It should be noted that—while we use ASCII values for readability in our example—input-to-state correspondence is also very applicable to binary formats.

```
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

Listing 2: Fuzzing problem (1): finding valid input to bypass magic bytes.

These constructs are hard to solve for feedback-driven fuzzers since they are very unlikely to guess a satisfying input; in this case the 64-bit input `MAGICHDR`. Existing approaches [16], [23], [34], [35], [38], [40] often use taint tracking and symbolic execution, both of which incur a certain performance overhead. An orthogonal approach are user-defined dictionaries [43] that represent expert knowledge about the program under test. Lastly, there are approaches that split multi-byte comparisons into many one-byte comparisons. Fuzzers are then able to solve individual bytes. The prime example is LAF-INTEL [2], which is very efficient at solving multi-byte compares, but needs source level access to modify the program. Another tool is STEELIX [31], which does not depend on access to the source code. Instead, it uses dynamic binary instrumentation to split large comparisons into smaller ones. Unfortunately, this approach has a large performance overhead. The authors of STEELIX reported that LAF-INTEL performs over 7 times as many executions per second.

TABLE I: Extracting the set of mutations from a comparison observed at run-time, using little-endian encoding.

C-Code	u64(*input*)	==	u64("MAGICHDR")
Input	"TestSeedInput"		
Observed (ASCII)	"deeStesT"	==	"RDHCIGAM"
Variations for < and > comparisons	"deeStesT" "deeStesT"		"RDHCIGAL" "RDHCIGAN"
Mutations after little-endian encoding	<"TestSeed" <"TestSeed" <"TestSeed"	↦ ↦ ↦	"MAGICHDR"> "LAGICHDR"> "NAGICHDR">

We propose the following lightweight approach based on input-to-state correspondence to handle magic bytes in a fully automated manner: we exploit the fact that values from the program state often directly correspond to parts of the input. Each time we encounter a new path, we hook all compare instructions and perform a single trace run. If we encounter a comparison with distinct arguments, we extract both arguments and create a custom mutation `<pattern ↦ repl>`, as we explain below. The different steps are illustrated in Table I.

i) Tracing. When we start fuzzing a new input (before entering the deterministic stage of KAFL), we perform a single run in which we hook all compare instructions and extract the arguments. This includes some instructions that are emitted by compilers to replace plain compare instructions or switch-case structures (by calculating offsets in jump tables). Additionally, we hook all call instructions, as functions might implement string comparisons and similar functionality. More details are given in Section IV.

Example 1. *Consider "*`TestSeedInput`*" as input for the code in Listing 2. The compare instruction checks if the first 8 bytes from the input, interpreted as an unsigned 64-bit value, are equal to the 64 bit unsigned interpretation of the string "*`MAGICHDR`*". As integers are typically encoded in little endian format, the ASCII representations of the final values used in the comparison are "*`deeStesT`*" and "*`RDHCIGAM`*".*

ii) Variations. At runtime, we do not know which flags are checked after the comparison; we cannot distinguish different comparison operations such as "lower than" and "equal to". Therefore, we apply some variations to the compared value such as addition and subtraction by one. As a side effect of this heuristic, we empirically found that this approach increases the probability of triggering off-by-one bugs.

Example 2. *In this case, we add and subtract* 1 *to/from "*`RDHCIGAM`*" and obtain "*`RDHCIGAL`*" and "*`RDHCIGAN`*".*

iii) Encodings. It is likely that the input has been processed in different ways before reaching the actual comparison. In order to handle the most common cases of input en-/decoding and to create more mutation candidates, we apply various different encodings to the mutation. Examples for these encodings are inverting zero extensions or endianness conversions.

Example 3. *We apply a little-endian encoding to our current mutations "*`RDHCIGAM`*", "*`RDHCIGAL`*" and obtain "*`MAGICHDR`*", "*`LAGICHDR`*" and "*`NAGICHDR`*".*

We observe that, generally, only a few primitive encoding schemes are required. By far the most common occurrence is a one-to-one mapping between input values and state values. In detail, the encodings we used in our experiments are:

- **Zero/Sign Extend**(n): the value is interpreted as little endian integer with zero or sign extension, leading bytes are stripped to produce a n byte version of the pattern, if applicable. When no size change takes place, this encoding is also called *plain* encoding.
- **Reverse**: all little endian encoding schemes also have a big endian equivalent.
- **C-String**: the value is a C string, and everything after the first 0 byte is removed.
- **Memory**(n): the value is treated as an argument to a function similar to `memcmp`. Consider only the first $n \in \{4, 5, \ldots, 32\}$ bytes.
- **ASCII**: the integer value is encoded as ASCII digits.

After manually evaluating the coverage produced by our fuzzer, we believe that the aforementioned set of encoding schemes covers the largest part of the common cases in real-world applications. In the rare cases where these encodings do not suffice, the set of encodings can also be considered as user input, similar to the dictionary in other fuzzing systems. In that case, the user can easily provide own, more specific encoding schemes. This step can be seen as a lightweight variant of a synthesis algorithm used to guess the symbolic state at the current position. In fact, this approach has one major advantage compared to other approaches for inferring how the input influences the state (such as symbolic execution or program synthesis); it is very easy to represent complex operations such as converting decimal ASCII numbers to integers. This is due to the fact that we only ever need to perform the encoding on concrete values instead of querying SMT solvers with symbolic values.

iv) Application. Finally and most importantly, we use the `pattern` of a mutation `<pattern ↦ repl>` to identify the parts of the input that are to be replace with the mutation `repl`. In contrast to other approaches such as ANGORA or STEELIX, we apply the whole pattern at once. This has two advantages: It works for atomic comparisons without further modification/hooking of the target and it drastically reduces the number of candidate positions at which to try the replacement.

Example 4. *Only the substring "*`TestSeed`*" of the input "*`TestSeedInput`*" is compared to "*`MAGICHDR`*". Therefore, we replace only this part with the generated mutations. This yields the new testcase "*`MAGICHDRInput`*", and by the variants introduced to solve inequalities: "*`LAGICHDRInput`*" and "*`NAGICHDRInput`*" (as well as potentially more inputs for other encoding schemes).*

v) Colorization. We found that the number of candidate positions to apply a patch is sometimes rather large. For example, the smallest valid ext4 file system image is 64 KB and mostly consists of long strings of zero bytes. Comparing a single zero value with some constant will yield more than $60,000$ possible positions. During our evaluation, we found that those comparisons occur rather frequently. Hence we devised an efficient procedure to increase the number of random bytes in the input. More entropy in the input reduces the space of possible positions. Using this "colored" copy of the input drastically reduces the number of candidate positions, usually by multiple orders of magnitude. After generating a colorized version, we only apply mutations where the `pattern` part was found at the same offset in both inputs. As a result, the remaining number of mutations to apply is reduced drastically. In our evaluation, we found that the number of mutations introduced by this approach is typically two orders of magnitude smaller than the number of deterministic mutations performed by AFL on the same input.

Example 5. *Assume that we are testing the input "*`ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ`*" in our running example. Amongst the mutations, we would find* `<ZZZZZZZZ ↦ MAGICHDR>`*. This mutation can be applied at many (24) different positions. Therefore, we try to replace as many characters as possible without changing the execution path. In this case, the colorized version might be any random string of bytes (such as "*`QYISLKFYDBYYSYWSIBSXEAXOKHNRUCYU`*"). Correspondingly, on a rerun, the same instruction would yield the mutation:* `<QYISLKFY ↦ MAGICHDR>`*, which is only applicable at the first position. Thus, we only produce one candidate at position 0.*

vi) Strings and Memory. Besides the aforementioned integer comparisons, programs often use functions to compare two strings or the content of byte arrays. Similarly, these tests often pose significant challenges to fuzzers. To overcome such constructs, we also hook all function calls. If the function takes at least two pointer arguments, we extract the first 128 bytes which are pointed to and treat them similarly to integers. However, we use a different set of encodings for memory content than for integers, most notably we either assume that only the first $n \in \{4, 5, \ldots, 32\}$ bytes are compared (functions similar to `memcmp`) or all bytes up to the first null byte (functions from the `strcmp` family).

vii) Input Specific Dictionary. Lastly, we add values that contain many consecutive non-zero or non-`0xff` bytes to a specific dictionary. The strings found this way will only be used during the havoc phase of the current input. This allows us to use values passed to functions whose inner workings are similar to comparisons while using non-trivial algorithms such as hashtable lookups. In a way, this is a much stronger version of the well-known trick to extract the output of the `strings` tool and use it as a dictionary for the fuzzing run since we

include dynamically computed strings, but not strings that are not relevant on this path.

B. Checksums

Another common challenge for fuzzers is to efficiently fuzz beyond checksums. A typical example for this challenge is depicted in Listing 3, which again represents an excerpt of our running example introduced in Listing 1.

```
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]=='R' && input[17]=='Q')
        bug(2);
```

Listing 3: Fuzzing problem (2): finding valid input to bypass checksums.

Existing approaches such as FLAYER [18], TAINTSCOPE [40] or T-FUZZ [34] all rely on the same idea: remove the hard checks and fix them later. TAINTSCOPE and T-FUZZ both detect critical checks automatically and, then, use symbolic execution to fix the checks once interesting behavior was found.

We propose replacing the taint tracking and symbolic execution used in TAINTSCOPE and T-FUZZ with the following procedure based on input-to-state correspondence: First, we identify comparisons that appear to be similar to checksum checks (e.g., one side is an input-to-state corresponding value, the other side changes regularly). Then, we replace the check with a comparison that *always* evaluates to true. Once the fuzzing process on this patched program produced a seemingly interesting path, we enter a *validation mode*. In this mode, we use the techniques described in the previous section to correct all patched comparisons. If this succeeds, we continue as before; otherwise, we learn that one comparison is not under our control and we remove the patch for this instruction to avoid performing unnecessary validation steps in the future.

Based on the idea of input-to-state correspondence, we are able to automatically infer which instructions to patch. Additionally, we can automatically repair the inputs without using human intervention or the complex primitives of taint tracking and symbolic execution. During feedback fuzzing, we fix any newly found input (in contrast to TAINTSCOPE and T-FUZZ). This ensures that no false positives are stored in the queue. As an additional benefit, this allows to share the queue with other tools. In the following, we discuss details of the process of selecting, patching, and validating suspected checksums with a focus on hashes and hash-like computations.

i) Identification. The first step happens during the processing of magic bytes, as described in Section III-A. This results in a list of comparisons and the values compared in all different colorized versions of the input. We use the following heuristic to filter the comparison instructions for interesting patch candidates related to checksums:

1) We are able to find the left-hand side of our mutation `pattern` in all inputs using the same encoding.
2) Neither argument is an immediate value.
3) `pattern` changes during the colorization phase (this is similar to the constraint that TAINTSCOPE uses: the value depends on many input bytes).

The intuition behind these checks is as follows: We observed that an instruction produced the mutation `<pattern` $\mapsto$ `repl>`. Assume `pattern` is a field from the input, and `repl` is the hash computed over a part of the input. In a checksum comparison, the left-hand side should always be part of the input and replacing large parts of the input with random values during the colorization should change the hash (and therefore `repl`). Similarly, both arguments cannot be immediate values if `pattern` is a value from the input and `repl` is the hash calculated over some part of the input. Obviously, this is an over-approximation and we sometimes find checks that are not part of an actual checksum. Therefore, this approach has a significant drawback: The removed instructions can be relevant bounds checks and removing them could introduce false positives (i.e., erroneous new coverage) or even cause the program to crash later on. We thus introduce a validation phase to weed out potential false positives and identify compare instructions that must not be patched. After the fuzzer finds a new input and before we store it in the queue, we try to fix all patched compare instructions. If we identify a patch that the fuzzer cannot fix automatically, we remove the patch immediately. Additionally, we discard the input before it ever reaches the queue. This ensures, that we do not waste time with patches that we cannot fix easily and that no false positives are produced: each input is validated with the unmodified executable.

ii) Patching. After we identified a set of suspected hash checks, we replace the instructions by patches which have the same side effects of a successful comparison. Obviously, this may lead to an undesired behavior: we might accidentally remove bound checks or make paths reachable that cannot be triggered without our patches. Nonetheless, we continue fuzzing with this newly patched binary because we will fix these problems later.

iii) Verification. After we performed the whole set of fuzzing stages on one input, we have a queue of preliminary results. Due to our patches, these inputs might not show the expected behavior on an unpatched target. During the verification phase, we try to fix these invalid inputs by applying input-to-state based mutations obtained from the patched instructions. Then, we execute the fixed inputs on the unpatched, real target. If they still trigger new coverage, the fixed input is written to the real queue. Otherwise, we discard the patch. After all preliminary inputs were processed this way, the preliminary queue is cleared for the next round.

IV. IMPLEMENTATION DETAILS

In the following, we provide a brief overview of REDQUEEN, the proof-of-concept implementation of our approach. We based our implementation of REDQUEEN on our fuzzer KAFL [37].

A. kAFL Fuzzer

KAFL is an OS-agnostic, AFL-inspired feedback-driven kernel fuzzer that uses the hardware-accelerated tracing feature Intel Processor Trace (Intel PT) to obtain coverage information without the need to instrument the target. It is build on top of modified versions of `KVM` and `QEMU` (namely `KVM-PT` and `QEMU-PT`) to execute arbitrary x86 operating systems in isolation. This way, the fuzzer can benefit from virtualization capabilities such as hardware-accelerated execution of code,

snapshots, and Intel PT tracing of the guest's code. KVM-PT enables Intel PT tracing of the guest and QEMU-PT decodes the generated traces into an AFL-compatible bitmap. To do so, QEMU-PT maintains a runtime disassembly of the target. This disassembly is perfect, in the sense that we disassemble each instruction which was executed according to the Intel PT trace. We use this disassembly to identify instructions to hook or to patch. This avoids problems which arise from patching code based on a static disassembly, which might misclassify some bytes. Both QEMU-PT and KVM-PT also provide custom hypercalls and direct memory access to the guest's memory to facilitate the necessary communication and data transfer with the target. The fuzzer logic is based on the AFL fuzzing loop (with an added `radamsa` [29] stage) and was re-implemented in Python. Hence, the fuzzing logic is independent from the target operating system. We also fixed a number of bugs in the decoding of Intel PT packets that removed a significant amount of non-determinism resulting from broken traces. In total we added and changed about 10k lines of code. A large part of these changes are not due to the techniques proposed in this paper. Most of these changes were performed to add support for ring 3 fuzzing, to provide VMI capabilities in KAFL and to fix bugs. Furthermore, these numbers also contain a significant amount of code used for evaluation and debugging purposes.

Our techniques require a few primitives: the ability to obtain a program trace, to inspect the program state at various breakpoints, and to patch instructions in memory. REDQUEEN still uses the architecture of KAFL and obtains coverage information in precisely the same manner. Additionally, it uses the VMI provided by KVM-PT and QEMU-PT to insert breakpoints and to inspect memory and register content during execution. In the following, we discuss how we implemented our techniques on top of KAFL.

B. Comparison Hooking

We rely on hardware-assisted virtual machine breakpoints to extract input-to-state correspondences. Each time the runtime disassembler encounters an interesting compare-like instruction, its address is stored such that a hook can be placed during the next REDQUEEN analysis phase. During the REDQUEEN phase, breakpoints are placed on all interesting instructions. When a breakpoint is hit, the arguments are extracted and saved to a buffer for later use by the fuzzing logic. Breakpoints are removed after they are hit a small number of times to limit the performance impact. Note, we do not only hook `cmp` instructions but also `call` instructions and subtractions. The former are used to identify string and memory compares, while subtractions are often emitted by compilers in place of `cmp` instructions to implement switch tables. To implement a subtraction, a compiler sometimes emits special `lea` or `add` instructions with a negative offset. If the first two arguments of a `call` instruction are valid pointers (according to various calling conventions), we assume the function to be a potential compare function and dump the first 128 bytes of memory for each argument.

C. Colorization

During the colorization step, we try to replace as many bytes with random values as possible, without changing the execution path (more precisely, the hash of the AFL bitmap). This increases the entropy in the input and, therefore, reduces the number of positions at which an observed pattern can be applied. This can be done using a binary search approach as shown in Algorithm 1 which will usually converge within a small number of executions (typically, in the order of a few hundred). The worst case would be comparable to one eighth of the number of bit-flips performed by AFL on each input. Additionally, in our implementation, we limited the search to a maximum of 1000 steps. This worked well even for the file system driver targets, with a minimum input size of 64 KB.

Algorithm 1: Algorithm for colorizing inputs to efficiently deal with a large number of candidate positions

```
   Data: input is the uncolored input
   Result: A version of input that has a significantly higher entropy
1  ranges ← (1...len(input))
2  original_hash ← get_bitmap_hash(input)
3  while rng = pop_biggest_range( ranges ) do
4      backup ← input[rng]
5      input[rng] ← random_bytes()
6      if original_hash ≠ get_bitmap_hash(input) then
7          add(ranges, (min(rng)...max(rng)/2))
8          add(ranges, (max(rng)/2 + 1...max(rng)))
9          input[rng] ← backup
```

D. Instruction Patching

Once the fuzzing logic has computed a list of candidate hash comparison instructions from the input-to-state correspondence data, we replace them with bogus compare instructions that always yields `true`. In our implementation, we use the instruction `cmp al,al` since it is the smallest compare instruction available on the x86 instruction set architecture. The remaining bytes of the original instruction are padded with `NOPs`. We use the `KVM` and `QEMU` debug facilities to apply these patches in memory inside of the VM. The normal fuzzing process is then continued with the patched VM. However, if patches are active, newly found paths are not immediately added to the queue. Sometimes even benign C code and common compilers emit assembly that jumps in the middle of instructions. In our case, this can be detected by the Intel Processor Tracing (PT) runtime disassembly. In other cases, techniques such as instruction punning [15] or even plain breakpoints might be used to avoid introducing unexpected crashes. However, it should be noted that even in relatively large target programs, we did not observe any such behavior, as the number of patched instructions is low (i.e., typically less than 5 in our empirical evaluation).

E. Input Validation and Fixing

We use Algorithm 2 to verify and fix preliminary results. This algorithm iteratively tries to fix all comparisons by repeatedly applying individual mutations and observing the resulting input-to-state correspondences. Note that there are situations in which there exists an order of comparisons, which we need to preserve while fixing the input. Typically, this is encountered if the header of a file format contains a checksum over the complete content of the file and some chunks inside the file are also protected by checksums. For example, the content of the IDAT chunk of PNG files is protected with a CRC-32 sum. If the content is `zlib` compressed, it is guarded

by another ADLER-32 checksum. In these situations, we have to fix the inner checksum first for the outer checksum to be calculated correctly. The order in which these checksums are checked is not obvious. However, it is more common for the outer checksum to be checked first. Therefore, we try to fix the last comparison first to avoid unnecessary work. In our experiments, this simple approach was sufficient. However, in general we cannot assume this to be the case. While performing a sequence of trial runs to fix all checksums in reverse order of occurrence, we observe which mutation influence which compare instructions. Thereby, we create a dependency graph of the different patched instructions. If the input from the first iteration is not valid, we use this dependency graph to perform a topological sort on the patches and to obtain a valid order. This allows us to apply another round of fixes to the input in the required sequence. If the final input does not exhibit the expected behavior on the unmodified executable, we remove the offending patches and discard the input from the preliminary queue.

Algorithm 2: Algorithm for fixing preliminary inputs

```
Data: input is the preliminary input with unsatisfied checksums
Result: Either (None, cmps) if the comparisons cmps could not be
        fixed or (in_f, None) if the new, fixed input in_f
        satisfies all comparisons
in_f←input
for cmp in reverse(patched_cmps) do
    in_f ← try_fix_value_for_cmp(cmp, in_f)
    if cmp ∈ get_broken_cmps(in_f) then
        return (None, {cmp})
    dependencies[cmp] ← get_cmps_influcenced()
if get_broken_cmps(in_f) ≠ ∅ then
    in_f←input
    ordered_cmps ← topological_sort(dependencies)
    for cmp in ordered_cmps do
        in_f ← try_fix_value_for_patch(cmp, in_f)
if get_broken_cmps(in_f) ≠ ∅ then
    return (None, get_broken_cmps(in_f))
return (in_f, None)
```

The routine `get_broken_cmps` used in Algorithm 2 returns a list of all patched compare instructions that are currently not satisfied, as well as the values that are being compared. The routine `try_fix_value_for_patch` tries to apply all mutations resulting from the patched instruction as described in Section III-A. Afterwards, this function checks if any of the mutations fixed the comparison. If such an input is found, it is returned. Otherwise, we learn that we cannot satisfy the compare instruction using our technique. The patch is removed from the list of patches used during further fuzzing runs. The routine `get_cmps_influenced` finds all compare instructions whose arguments were influenced by the last fix applied to the input. This allows us to construct the ordering of compare instructions that we use later, if required.

Example 6. *Consider Bug 2 of our running example shown in Listing 1. After we removed the two checksum checks, the fuzzer finds the preliminary input "*`01234567abcdefghRQ`*". Tracing yields the following results:* `p1 := <01234567 ↦ \xc7\x03\0\0\0\0\0\0>` *and* `p2 := <abcdefgh ↦ \xa3\0\0\0\0\0\0\0>`*. If we were to apply both mutations at the same time, the second mutation would invalidate the sum of the first one. Therefore, we first try to fix the second patch (*`p2`*) and note that* `p1` *is influenced. After fixing* `p2`*, we obtain the input: "*`01234567\xa3\0\0\0\0\0\0\0RQ`*". Changing the inner checksum of the input, also changes the expected value for* `p1`*. The new mutation for* `p1` *is now* `<01234567 ↦ \x46\x01\0\0\0\0\0\0>`*. Then we apply the first patch* `p1`*. This time we do not disturb any of the other patches. The final input is: "*`\x46\x01\0\0\0\0\0\0\xa3\0\0\0\0\0\0\0RQ`*". As all patched constraints were satisfied by this input, we perform one last run without patches to ensure that the input behaves as expected. This input does indeed trigger Bug 2 and is moved from the preliminary queue to the real queue. In this example we did not need the topological sort operation to order the checks, as we guessed one correct order to fix the patches.*

F. Linux User Space Application Loader for KAFL

We extended KAFL by a Linux ring 3 loader to evaluate against other user space fuzzers and to demonstrate that our approach is generic and robust. This loader re-implements the AFL fork server. Since we are targeting binaries, we use `LD_PRELOAD` to inject the fork server functionality into the start-up routine of the target. Communication with the fuzzing logic is performed with custom KAFL hypercalls triggered by the injected start-up routine. Also, to support ring 3 tracing in `KVM-PT`, we set the `User` bit in the model specific register `IA32_RTIT_CTL_MSR`. Since the original KAFL was designed to be a kernel fuzzer, it sets only `IA32_RTIT_CTL_MSR.OS` to enable ring 0 tracing. Additionally, the original fuzzer was only intended to fuzz 64-bit operating systems. However, since some of the `CGC` binaries can only be compiled for 32-bit targets, we extended the fuzzer to support 32-bit targets. Hence, we added 32-bit mode disassembling to `QEMU-PT` to support decoding of 32-bit mode Intel PT trace data.

V. Evaluation

We evaluated our prototype implementation of REDQUEEN as described above to answer the following research questions:

- **RQ 1.** Are input-to-state correspondence-based techniques general enough to work across a diverse set of targets and environments?
- **RQ 2.** How do the results of our input-to-state correspondence-based techniques compare to other, more complicated techniques such as approaches based on taint tracking or symbolic execution?
- **RQ 3.** What improvements do our input-to-state correspondence-based techniques provide in real-world fuzzing scenarios?

To answer these questions, our evaluation is divided into three parts: First, we perform a comparative evaluation on two synthetic test sets (LAVA-M and `CGC`) and a real-world test set (GNU `binutils`). The experiments demonstrate that our combination of methods outperforms all other current approaches by a significant margin. Second, we demonstrate that our tool is able to find novel bugs in various well-tested

software packages, running in very different environments. In total, we found 10 bugs in 2 different Linux file system drivers and 55 bugs in 16 user-space programs and software libraries. So far, we obtained 16 CVEs, 4 are still pending. Finally, we measure the efficiency and effectiveness of our technique as compared to the other mutations performed by KAFL. We also test the influence of the individual techniques introduced in this paper, using a case study based on a small statically linked PNG library because this file format uses nested checksums. We demonstrate that our approach enables us to overcome fuzzing roadblocks that previously required dictionaries and manual removal of hash checks.

A. Evaluation Methods

All experiments were performed on systems running Ubuntu Server 16.04.2 LTS with an Intel i7-6700 processor (4 cores) and 24 GB of RAM. We configured all tools to use one fuzzing process to ensure comparability with tools such as VUZZER which are not multi-threaded. No experiments contain changes made specifically for certain targets. Unless stated otherwise, we used an uninformed, generic seed consisting of different characters from the printable ASCII set: `"ABC...XYZabc...xyz012...789!¨$...~+*"`. Since there exist various definitions of a basic block and tools such as LAF-INTEL dramatically change the number of basic blocks in a binary, we always measure the coverage produced by each fuzzer on the same uninstrumented binary. As such, the numbers of basic blocks uncovered may not match the numbers reported in other papers but are guaranteed to be consistent within our experiments. All experiments were conducted multiple times. In each case, we report the median number of basic blocks found at any time, as well as the 60% confidence intervals. Recently, much focus was placed on uncovering "deep" bugs [34]. While the exact definition of "deep" remains somewhat elusive, we use the following definition as a work in progress: A bug is considered "deep" if it is hard to reach from the given seed inputs. We, therefore, consider the ability to find new code coverage a good proxy for the ability to find "deep" bugs—a property that is much harder to qualify. Lastly, since the nomenclature varies across related research, we use "crash" to describe any input that made the target application crash. We never talk about the number of "uniqe" crashes found, as this metric is highly unreliable, as different tools report drastically different and often inflated numbers of crashes. Instead we say "*we were able to crash*" to denote that at least one crash was found without further triage. We use the term *bug* to describe manually verified and triaged bugs with disjoint root causes in the application. A bug is not necessarily exploitable. Lastly, in some cases, we obtained CVE numbers which we count and list individually.

B. LAVA-M

LAVA-M [17] is a synthetic set of bugs inserted in hard-to-reach places in real-world binaries from the GNU coreutils suite. It is commonly used to evaluate the efficiency of modern fuzzers. We first describe the experiments performed on LAVA-M by other publications and then compare them with our results, a summary is shown in Table II.

In the original paper of LAVA-M, the authors perform two experiments: one using an unspecified FUZZER and an unspecified symbolic execution tool SES (both are undisclosed "state-of-the-art, high-profile tools" [17]). Both tools were run against the corpus for 5 hours on an unspecified machine. All experiments performed by the STEELIX authors were run on a machine with 8 Intel(R) Xeon(R) CPU E5-1650 v3 cores and 8 GB of RAM, running 32-bit Ubuntu 16.04 using one thread. Similarly, Sanjay et al. performed their experiments with VUZZER [35] using a 5-hour run on an unspecified machine with 4 GB of RAM on a single core. We performed a similar experiment where each target was executed for 5 hours. We used the first few bytes of the seed provided by the original authors as well as our usual uninformed seeds. Figure 1 displays the results (5 runs of 5 hours each, median and the 60-percent confidence interval of the number of bugs found over time, in percent of the number of bugs intended by the authors of LAVA-M). In all cases, we found more bugs than the original authors of LAVA-M intended. Also, in all cases, we needed much less than the full 5 hours to find all bugs. In the median case, it took less than 5 minutes to find more bugs on `who` and `base64`. After 15 minutes, `uniq` was fully solved. Lastly, `md5sum` was the slowest target, taking 25 minutes. The reason why `md5sum` was slower than the other targets is that we were using a busybox environment in which all files in `/bin/` link to the same monolithic binary, significantly slowing down the md5 calculation.

We found all bugs inserted and listed, except for two bugs in `who`. More importantly, we found a significant number of bugs (337) considered *unreachable* by the original authors. The only other fuzzer that managed to find some of these unlisted bugs was ANGORA. However, we managed to find over 50% more bugs than ANGORA and more than *three* times as many unlisted bugs. Due to the high number of unlisted bugs, we contacted one of the authors of LAVA-M who confirmed our findings from the inputs we provided.

From the fact that the difference between informed seed inputs and uninformed seed inputs is marginal and the fact that we outperformed the other current state-of-the-art tools by factors ranging from *8x to 26x*, we conclude that the LAVA-M data set favors the hypothesis that our approach outperforms state-of-the-art approaches and is able to uncover deep bugs far from the provided seed inputs. In fact, we are outperforming all current approaches, even if no seed is given to our tool. However, it should be noted that the bugs inserted into the LAVA-M data set are artificial and do not represent real-world vulnerabilities.

C. Cyber Grand Challenge

The targets used in DARPA's CGC are another widely used test set to evaluate fuzzers. Similarly to the LAVA-M corpus, we will now describe the experimental setups used by various other fuzzers on these targets. Then, we will compare their results with our results.

STEELIX only tested a subset of eight CGC binaries and compared their results against AFL-DYNINST, an AFL version that uses dynamic binary instrumentation and is significantly slower than the original AFL. Both tools were allowed to fuzz for three hours. STEELIX was able to find one crash that AFL could not find. In contrast, we were able to find the crash in *less than 30 seconds* with the seed used by STEELIX.

TABLE II: Listed and (+unlisted bugs) found after 5 hours of fuzzing on LAVA-M (numbers taken from the corresponding papers).

Program	Listed Bugs	FUZZER	SES	VUzzer	STEELIX	T-Fuzz	Angora		Redqueen	
uniq	28	7	0	27	7	26	28	(+ 1)	28	(+ 1)
base64	44	7	9	17	43	43	44	(+ 4)	44	(+ 4)
md5sum	57	2	0	-	28	49	57	(+ 0)	57	(+ 4)
who	2136	0	18	50	194	63	1443	(+ 98)	2134	(+ 328)

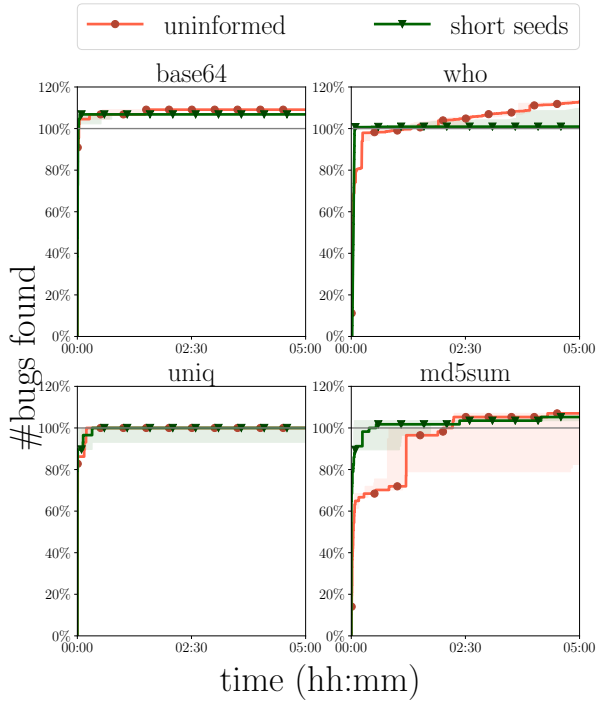

Fig. 1: Evaluating Lava-M on informed and uninformed seeds.

VUzzer tested a subset of 63 binaries, filtering binaries that use floating-point instructions (which VUzzer's taint tracking cannot handle), IPC, or binaries that are easily put into an infinite loop. The authors kindly provided us with the exact set of targets they used and the set of targets in which they were able to uncover crashes. Unfortunately, they were not able to provide us with the exact seeds used. We re-created a similar set of seeds with the same method. Some inputs were generated by the poller scripts used in CGC and some inputs were taken from the original DARPA repository. However, we were unable to create seeds for nine of the challenges. Consequently, we excluded them from our evaluation. VUzzer was able to find crashes in two of these excluded binaries. The experiments with VUzzer were performed on a host with two unspecified 32-bit Intel CPUs and 4 GB of memory inside the original DECREE VM. Both VUzzer as well as AFL-PIN were given 6 hours of fuzzing time. They were able to find 29 crashes in 27 binaries. In this setup, AFL-PIN was able to uncover 23 crashes. A significant number of the target binaries runs in a busy loop waiting for a specific "quit" command. This leads to a large number of timeouts, greatly reducing the fuzzing performance. While such a loop would be easy to detect in a limited setting such as CGC, this optimization would be out of scope for a more general fuzzer. To mitigate this problem, the VUzzer authors manually ensured that VUzzer always appends the correct "quit" command.

We run a similar set of experiments with a few differences: In contrast to the authors of VUzzer, we used the multi-OS version released by Trail of Bits [3]. Therefore, some of the comparisons might be biased due to slight differences. Under these conditions, we were able to trigger crashes in 31 binaries on the first try with Redqueen, while VUzzer was able to trigger 25 crashes.

Since these results were produced without adapting Redqueen to CGC at all, this experiment serves as a validation set to ensure we did not overfit our fuzzing process to any specific target. Since the AFL fuzzing model is primarily aiming at binary input formats—CGC contains a large number of interactive line base formats—we performed another experiment: we added some mutations relevant for text-based protocols (mostly splicing lines as well as adding random lines). We also used 4 cores each for 2 hours instead of the usual configuration to speed up experiments. In this experiment, we were able to uncover 9 additional crashing inputs. In total, we were able to crash 40 out of the 54 targets. VUzzer was able to find crashes in 5 binaries which we could not crash. In at least one case this was due to heavy timeouts, which VUzzer avoided by adapting their fuzzer to the specific targets as described earlier (which we did not). On the other hand, we crashed 19 binaries that VUzzer was unable to crash. This suggests that we are able to outperform both VUzzer by 60% as well as AFL-PIN by 73% on this data set.

Unfortunately, the authors of T-Fuzz did not disclose the full results of their CGC experiments. However, they state that, on the subset of binaries that VUzzer used, they found 29 crashes in the 6 hours allocated—significantly less then the 40 crashes that our approach managed to find.

Driller was evaluated on the whole set of CGC binaries. The experiment used a cluster of unspecified AMD64 CPUs with four cores per target. The concolic execution processing was outsourced to another set of 64 concolic execution nodes. Each concolic execution was limited to 4 GB of memory and every target was fuzzed for 24 hours. Fuzzing with AFL was able to uncover crashes in 68 targets. Driller was able to increase this number to 77. It should be noted that, as a participant in the CGC competition, Driller is highly optimized to the CGC dataset. Yet, the improvements over baseline fuzzing with AFL were only in the order of 14 %. Our results on the dataset of VUzzer compared to AFL-PIN (73%) suggest that we might be able to see improvements similar to Driller if we were to implement the various DECREE specific mechanisms such as the custom IPC.

D. Real-World Applications

We also evaluated REDQUEEN on several real-world applications. Generally speaking, we found all the types of bugs fuzzers usually find: out-of-bound read/write accesses to memory, resource exhaustion (both, time and memory), memory leaks, stack overflows, division by zeros, assertions, use-after-free, use of uninitialized values, and so on. For the purpose of this evaluation, we disregarded the following bug classes, as they were far too numerous for manual triage and of little importance for security research: memory leaks, resource exhaustion, use of uninitialized values (unless they lead to more severe consequences) and stack overflows.

binutils. To estimate the ability to uncover deep bugs in hard-to-reach code, we measured the coverage produced by various tools on real-world binaries from the `binutils` collection to expand our evaluation from synthetic test cases to more realistic tests. We use code coverage as a proxy metric for the ability to uncover deep bugs, as the number of bugs found is very hard to establish. To properly compare the number of bugs found across tools, many thousands of crashes would have to be investigated. In many cases, most tools were unable to uncover a single crash, leading to non-descriptive experiments. We regard code coverage as a very good proxy, as no fuzzer can find bugs in code that is not covered. As test suite, we picked all eight programs from the `binutils` collection, which are processing one file without modifying it. Unfortunately, we cannot evaluate against DRILLER since it is only applicable to `DECREE` binaries. Also, since both ANGORA and T-FUZZ are not yet available, we cannot run our own experiments to compare against them. We, therefore, choose VUZZER as one of the few available academic state-of-the-art fuzzers, as well as AFL with the LAF-INTEL and AFLFAST extensions. Notably, both LAF-INTEL and VUZZER explicitly aim to overcome magic bytes and other fuzzing roadblocks, similar to our tool. In all cases, we started with a single uniformed seed to measure the ability to solve roadblocks. KLEE does not produce test files as it uncovers new states. Instead, it only produces a test case after the state has been fully explored or after the timeout has been reached. Therefore, it is not uncommon that KLEE writes only a very small number of test cases while running. After reaching the timeout, it starts to solve all remaining states to produce more coverage, and, in some cases, it can spend many hours of additional computation time to produce the actual test cases. We, therefore, gave KLEE a 7 hour window to evaluate symbolic states plus 3 hours for test case production. We forcefully terminated KLEE after a total of 10 hours. The same architecture makes it very hard to determine the exact time when a new input was found. For plotting purposes, we assumed that inputs are found at a constant rate. This assumption is obviously wrong, as we would expect far more inputs to be found during the early hours. However, since we primarily compare the final number of basic blocks covered, the exact shape of the curve in the plot is not overly relevant. VUZZER is unable to fuzz targets that expect their input on **`stdin`**, and, therefore, we do not have VUZZER results for the **`cxxfilt`** target. To represent other Intel PT based fuzzers, we included HONGGFUZZ [6] running in Intel PT mode and the original KAFL with our Ring 3 extension.

The results of five 10 hour runs can be seen in Figure 2.

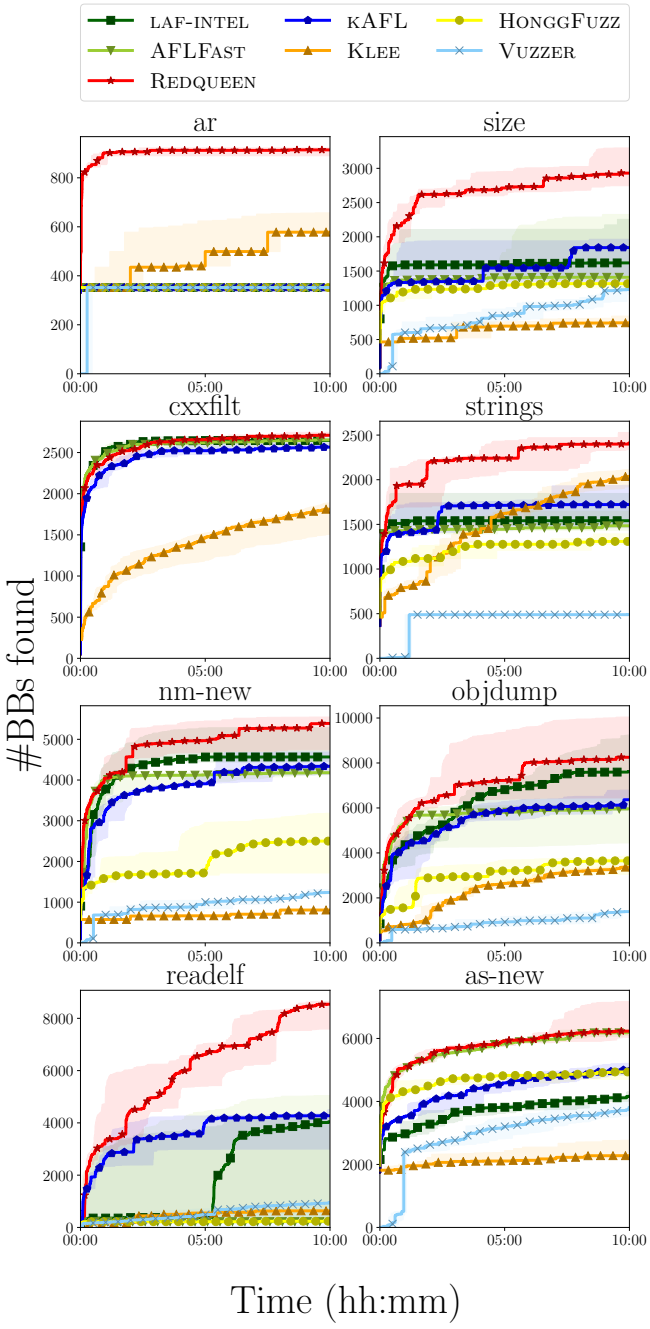

Fig. 2: The coverage (in basic blocks) produced by various tools over 5 runs at 10 h each on the `binutils`. Displayed are the median and the 60 % intervals.

Notably, in all cases, REDQUEEN is able to trigger the most coverage. To ensure these findings are not the result of random variation, we employed a Mann-Whitney U test on the number of basic blocks found, as recommended by Arcuri et al. [7] for evaluation of randomized algorithms. The results are displayed in Table III. It can be seen that, in nearly all cases, the observed differences are significant at $p < 0.05$. Interestingly, VUZZER finds very few new basic blocks. We investigated this behavior to ensure we are not misusing VUZZER. We contacted the authors, who reproduced our experiments and found very similar results. In the resulting conversation, it became clear that VUZZER strongly relies on the assumption that there is only one valid input format. Since most `binutils` programs

TABLE III: p-values of the Mann-Whitney U test on the number of basic blocks found in 10h. Nearly all results are statistically significant ($p < 0.05$).

Target	LAF-INTEL	AFLFAST	KAFL	KLEE	HONGG	VUZZER
ar	**0.004**	**0.004**	**0.004**	**0.005**	**0.004**	**0.004**
cxxfilt	**0.014**	**0.006**	**0.006**	**0.006**	-	-
nm-new	**0.018**	**0.006**	**0.006**	**0.004**	**0.006**	**0.006**
readelf	**0.006**	**0.006**	**0.006**	**0.006**	**0.005**	**0.006**
size	**0.006**	**0.006**	**0.006**	**0.006**	**0.006**	**0.006**
strings	**0.006**	**0.006**	**0.006**	**0.006**	**0.006**	**0.004**
objdump	0.265	**0.006**	**0.018**	**0.006**	**0.006**	**0.006**
as-new	**0.006**	0.500	**0.006**	**0.006**	**0.006**	**0.006**

read a vast amount of different formats, VUZZER excludes a significant number of paths from the search. In addition, VUZZER expects valid inputs to detect error paths. However, our uninformed inputs did not seem to limit VUZZER severely as it still was able to find valid `ELF` files immediately (e.g., it did not discard the interesting paths as "uninteresting error cases"). Using REDQUEEN, we found and reported bugs in the following binutils: `ld-new`, `as-new`, `gprof`, `nm-new`, `cxxfilt` and `objdump`. As it turned out, the bugs found in `cxxfilt`, `nm-new` and `objdump` were all instances of the same bug in the shared library that demangles C++ symbols.

We found that REDQUEEN, LAF-INTEL and AFLFAST all reported a significant higher number of crashes. However, most of these "crashes" were inputs that exceeded the memory limits set by the fuzzers. We manually validated the crashes found in `objdump` and verified that neither AFLFAST nor LAF-INTEL found any real bug, while some of the crashes identified by REDQUEEN were indeed novel bugs. This indicates that the common practice of reporting the number of found crashes can be a misleading indication and a proper bug triage is necessary.

Case Study: `objdump`. A surprising observation was that we were able to solve a complex constraints in `objdump` that used the hashtable lookup function `bfd_get_section_by_name(bfd* obj, char* name)` to test if a section with the given name is present in the input object file `obj`: Since the function was given two pointer arguments, REDQUEEN automatically extracted the name and added it to the dynamic dictionary. The havoc stage was later able to insert it in the right position. Similarly, the reason that all other fuzzers show a lower performance on `ar` is that `ar` uses a call to `strncmp` to check that the first few bytes are "`!<thin>`" or "`!<bout>`". Usually, LAF-INTEL is able to split such string compares. However, LAF-INTEL only considers the functions `strcmp` and `memcmp` but not the size-restricted versions. This is a nice example where more precise approaches need a very detailed environment model. Both results demonstrate how the simple over-approximations we use are actually helpful in real-world applications.

Other Targets. To ensure that our tool is able to find novel bugs, we evaluated the bug finding ability on a diverse set of real-world targets. The results can be seen in Table IV. In all cases, we manually triaged and reported the bugs found. We started with targets commonly used in other papers, namely `libtiff` (`tiff2ps`) [31], [34], [40], `imagemagick` [34], [40] and `jhead` [16], [28], each in the most recent version of `Ubuntu` (16.04 LTS) where these previously reported bugs should be fixed. Even though these tools have been exhaustively tested previously, we found novel bugs which we triaged and reported. We also evaluated on a set of media file format based tools: `sam2p`, `wine` and `fdk-aac` (a part of `ffmpg`). In the spirit of the original KAFL paper, we used REDQUEEN to target two file system drivers (specifically `hfs.ko` and `ntfs.ko`) from the most recent `Ubuntu` release (16.04 LTS). In both cases, we found and reported multiple memory corruptions. Lastly, in addition to the (mostly) binary focused targets we evaluated so far, we also tested multiple well known text-based targets: `mruby`, `perl`, `bash` and `libxml2`. In nearly all of the targets we used our uninformaed seed to find the bugs. This experiment demonstrates that our approach is applicable not only to user-space code but also to kernel-level code.

TABLE IV: Reported bugs found by REDQUEEN

Application	Version	Bugs	CVEs
`ld-new`	`binuitils-2.30-15ubuntu1`	3	-
`as-new`	`binuitils-2.30-15ubuntu1`	8	-
`gprof`	`binuitils-2.30-15ubuntu1`	2	-
`nm-new`	`binuitils-2.30-15ubuntu1`	1	1
`cxxfilt`	`binuitils-2.30-15ubuntu1`	1	1
`objdump`	`binuitils-2.30-15ubuntu1`	11	4
`tiff2ps`	`libtiff-4.0.9`	4	-
`jhead`	`3.00-6`	13	1
`fdk-acc`	`v0.1.6`	2	-[a]
`ImageMagick`	`7.0.7-29`	3	-[a]
`wine`	`wine-2.0.2-2ubuntu1`	3	2
`mruby`	`Commit 51614bbddb5...`	1	1
`sam2p`	`Commit 5ed411dd145...`	1	-[a]
`bash`	`Commit 64447609994...`	2	1
`libxml2`	`Commit 35e83488505...`	1	1
`perl`	`5.26.1...`	1	-[a]
`hfs.ko`[b]	`kernel-4.15.0-15.16`	1	1
`ntfs.ko`	`kernel-4.15.0-15.16`	9	3
Total		**65**	**16**

[a] CVE assignment pending.
[b] Simultaneously found and reported by `syzbot`.

Case Study: `mruby`. One very interesting vulnerability we found was based on an integer overflow in `mruby`. When resizing a string, the next power of two was chosen as new allocation size. The computation could overflow, resulting in a negative size. This was prevented by an additional sign check after the computation. The compiler realized that powers of two are always positive and, since signed integer overflows are undefined behaviour in `C`, removed the check. Therefore, we were able to produce strings with negative length. Since the new length was smaller than the old length, no allocation took place, but the strings length was updated to a negative value. The resulting string then spans the whole memory range. This behavior was only present in uninstrumented executables compiled with `gcc` and optimization level 2. This bug very nicely demonstrates why it is important to have techniques that work effectively on binary-only targets. We strongly suspect that the overflowing integer (a 64-bit values with 19 digits) was found by the ASCII integer encoding.

E. Baseline Evaluation: PNG File Format

We performed the following experiments to demonstrate that the improvements gained in our experiments are indeed due to the proposed techniques. We use the `lodepng` library, a small library that can easily be linked statically and which facilitates the loading of PNG files. The PNG format is an interesting case study for common fuzzing roadblocks because it is based on a list of chunks. Each chunk starts with a header (identified by a 4-byte magic value) and contains a `CRC32` checksum over the content. The content of the IDAT

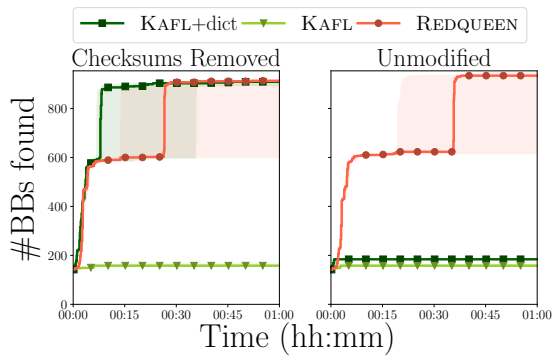

Fig. 3: Evaluating the impact of REDQUEEN vs KAFL.

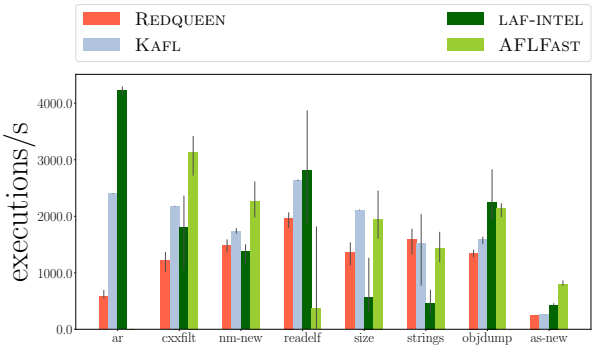

Fig. 4: Evaluating the execution speed on our `binutils` targets.

chunk is the `zlib` compressed pixel data, together with another `Adler-32` checksum. In each experiment, we performed 15 runs of one hour each and measured the number of basic blocks found over the time. The results of the experiments are displayed in Figure 3. The figure contains the median number of basic blocks found at each point in time, as well as the confidence intervals.

First, we validate that magic bytes are successfully solved by REDQUEEN but not by KAFL. To do so, we disabled the two checksum checks in the binary and compare the results for KAFL, KAFL with a dictionary containing all relevant magic bytes, and REDQUEEN. The results of this experiment are displayed in the "Checksums Removed" configuration of Figure 3. It can be seen that the dictionary massively increases the coverage produced by KAFL compared to the baseline. However even without a dictionary provided, REDQUEEN is able to achieve exactly the same coverage—even though it takes a little more time to do so. For the next experiment, we use an unmodified target to demonstrate that REDQUEEN is able to overcome checksums. Again, KAFL without REDQUEEN is evaluated both, with and without a dictionary. Since KAFL is unable to overcome checksums, it only finds a very small number of paths, regardless of the dictionary. REDQUEEN, on the other hand, is able to identify and solve the relevant checksums, and thus achieves as much coverage as in the previous experiment. In fact, it even finds a few more basic blocks (the checksum calculation which is now active). This case study demonstrates that we are able to overcome checksum; furthermore, it demonstrates how REDQUEEN is able to deal with these roadblocks in an automated manner. For vanilla KAFL, we needed to disable checksums (a task not easily achieved on closed-source targets) and provided a custom dictionary to obtain interesting coverage. This experiment indicate that our techniques are as effective as manually finding a good dictionary and removing checksums.

F. Performance

In this experiment, we measure the efficiency and effectiveness of our approach. To measure the efficiency, we compare the overall number of executions per second achieved by REDQUEEN, KAFL, LAF-INTEL, and AFLFAST. We obtain a measure for effectiveness by considering the percentage of inputs found by the different mutation engines used in REDQUEEN. Lastly, we evaluated the prevalence of the proposed encoding schemes. In all cases, we use the data from the experiment on the `binutils` suite in Section V-D.

TABLE V: Number of inputs that trigger paths with new coverage per time spent using different techniques. The time for our input-to-state based mutator includes tracing, colorization, and execution of all proposed inputs. In most cases, input-to-state correspondence produces significantly more inputs per time than any other mutation method used.

Mutation	Target	#Inputs	Time Spend (min)	#Inputs/min
Input-To-State	`ar`	67	0.7	**98.0**
	`size`	465	7.9	**58.7**
	`cxxfilt`	309	11.2	27.7
	`strings`	315	8.0	**39.4**
	`nm-new`	1016	24.3	**41.8**
	`objdump`	1189	30.7	**38.8**
	`readelf`	1616	21.1	**76.5**
	`as-new`	717	31.6	22.7
Deterministic Stages	`ar`	59	35.1	1.7
	`size`	578	148.9	3.9
	`cxxfilt`	706	241.2	2.9
	`strings`	413	287.3	1.4
	`nm-new`	1767	179.6	9.8
	`objdump`	2385	242.7	9.8
	`readelf`	2929	331.3	8.8
	`as-new`	884	437.3	2.0
Havoc	`ar`	32	70.9	0.5
	`size`	439	149.2	2.9
	`cxxfilt`	4584	133.1	**34.4**
	`strings`	381	128.2	3.0
	`nm-new`	1748	167.9	10.4
	`objdump`	2619	164.0	16.0
	`readelf`	1161	113.7	10.2
	`as-new`	1769	75.4	**23.5**
Splice	`ar`	18	64.5	0.3
	`size`	86	119.5	0.7
	`cxxfilt`	1140	152.1	7.5
	`strings`	95	115.1	0.8
	`nm-new`	780	117.3	6.7
	`objdump`	335	68.6	4.9
	`readelf`	251	61.5	4.1
	`as-new`	510	31.1	16.4
Radamsa	`ar`	0	422.2	0.0
	`size`	2	166.0	0.0
	`cxxfilt`	237	47.2	5.0
	`strings`	18	43.3	0.4
	`nm-new`	5	99.4	0.1
	`objdump`	3	81.0	0.0
	`readelf`	2	60.8	0.0
	`as-new`	69	10.2	6.8

The overall fuzzing performance can be seen in Figure 4. The bars represent the average number of executions performed per second. The usual performance impact for KAFL and REDQUEEN compared to LAF-INTEL and AFLFAST typically in the range of 25-50 %. This is due to the fact that KAFL does not use the fast compiler-based instrumentation and is working on binary code. Nonetheless, to the best of our knowledge, REDQUEEN is by far the fastest binary-only fuzzer.

TABLE VI: Percentage of the paths found by different encoding schemes.

Encoding	Plain	Reverse	Total
Zero Ext	38.40 %	6.82 %	45.22 %
Sign Ext	22.65 %	6.61 %	29.27 %
Plain	4.42 %	11.64 %	16.06 %
Mem	5.16 %	-	5.16 %
C-Str	2.42 %	-	2.42 %
ASCII	1.21 %	-	1.21 %

In one cases, REDQUEEN performs slightly better than KAFL. While this result is counter-intuitive (after all, we perform additional work), there is a good explanation for this behavior: On real-world targets, the number of executions achieved per second is heavily dependent on the inputs in the queue. If the queue is filled with slow inputs, the performance drops. When KAFL is unable to overcome a roadblock, the only new inputs are inputs that perform more loop iterations. As it can be seen in Figure 4, the impact of the REDQUEEN extension on the number of executions per second is rather low and sometimes it even increases the number of executions performed. Even though the breakpoint based tracing is slow (often by a factor of 50x or more), it hardly influences the fuzzing performance as the REDQUEEN phase is only triggered once per input in the queue. The measurements for the effectiveness are show in Table V. All techniques are used by REDQUEEN on each input, and the number of of new inputs found by each technique are noted. It can be seen that input-to-state correspondence based mutations are often finding as many, or more, new inputs than the other phases while using significantly less time. This shows that our techniques are useful far beyond the archetypical roadblocks: They help finding a significant number of paths faster than other mutation strategies. Lastly, we evaluated the prevalence of different encoding schemes by counting the percentage of paths found using each encoding scheme. The results are shown in Table VI. It can be seen that nearly all the improvements that we made are due to what we consider one-to-one correspondences. Due to the last two experiments, we consider input-to-state-based mutations a highly general, effective, and efficient scheme that can be used to significantly improve the performance of fuzzers.

G. High-Level Summary

After showing that we are able to substantially increase the coverage produced on three large test corpora and to find novel bugs in both kernel- and user-space software, we conclude that **RQ 1.** holds and input-to-state correspondence based techniques are useful for fuzzing purposes. After comparing the results of state-of-the-art tools on the same test sets, we feel confident to answer **RQ 2.**: we are able to outperform current symbolic execution and taint tracking based approaches, represented by VUZZER, ANGORA, T-FUZZ and KLEE, when measuring the ability to uncover new behaviour and bugs. Lastly, we demonstrate that—even in a binary-only scenario—our input-to-state correspondence based techniques are powerful enough to compete with other approaches, even if hash checks are removed for the competing tools and they are provided with a proper dictionary. This answers **RQ 3.** In summary, we feel confident that our input-to-state correspondence method significantly improves coverage-based feedback fuzzing.

VI. LIMITATIONS

Our approach is applicable without access to the source code and without knowledge of the environment. Yet, it outperforms state-of-the-art fuzzers even when they have access to the source code. However, this does not mean that more complex approaches are useless. Instead, we strongly believe these approaches are useful in certain cases where our approach does not offer any advantages. We encountered some examples for such cases while manually inspecting the different targets discussed in this paper: the PNG file format, contains compressed data which the decoder decompresses during execution. In some programs of the `binutils` test set, a string from the input is used to index a hash map, returning an integer that is used thereafter. Lastly, the `base64` utility from the LAVA-M data set applies base64 decoding. Our current implementation does not provide an encoding scheme for base64. In all of these cases the input does not correspond to the state after the transformation took place. Therefore our approach is unable to efficiently solve constraints that occur afterwards. We believe that the first two cases (compression and hash maps) are also hard for concolic execution based approaches. Most concolic engines cannot efficiently handle cases where one has to use different paths to change values. In the case of base64, we could easily add another encoder. However, as our evaluation shows, these cases are rare. Consequently, tools that are able to solve these cases struggle with more common problems, such as path explosion or poor performance on complex, real-world targets. We believe that it would be beneficial to use our lightweight approach as a first step where possible, and than solve the remaining challenges using complex approaches.

VII. CONCLUSION

In this paper, we presented and evaluated methods based on input-to-state correspondence to improve fuzzing. We have shown that it is possible to significantly improve the coverage in binary-only targets by solving magic-bytes comparisons and checksum checks. While our approach is not as grounded in formalism as other approaches based on symbolic execution or taint tracking, we believe that it is very much in the spirit of AFL: it is fast, lightweight, and—most importantly—robust. Even if some parts of the program are very hard to analyze, our approach remains applicable and effective in other parts of the target. We believe that our work shows that we can significantly improve the performance of fuzzers without falling back to fragile and complex methods.

ACKNOWLEDGMENTS

This work was supported by Intel as part of the Intel Collaborative Research Institute “Collaborative Autonomous & Resilient Systems” (ICRI-CARS). The research leading to these results has received funding from the European Unions Horizon 2020 Research and Innovation Programme under Grant Agreement No. 786669. The content of this document reflect the views only of their authors. The European Commission/Research Executive Agency are not responsible for any use that may be made of the information it contains. Finally, we would like to thank Ali Abbasi, Joel Frank, Emre Güler and Christine Utz for their valuable feedback.

REFERENCES

[1] Announcing OSS-Fuzz: Continuous fuzzing for open source software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

[2] Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/. Accessed: 2018-08-07.

[3] Darpa challenge binaries on linux, os x, and windows. https://github.com/trailofbits/cb-multios. Accessed: 2018-08-07.

[4] Peach. http://www.peachfuzzer.com/. Accessed: 2018-08-07.

[5] Project Triforce: Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/.

[6] Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: 2018-08-07.

[7] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[8] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[9] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[11] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[13] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy*, 2012.

[14] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[15] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. Instruction punning: Lightweight instrumentation for x86-64. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[16] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

[17] Brendan Dolan, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy*, 2016.

[18] Will Drewry and Tavis Ormandy. Flayer: exposing application internals. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2007.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy*, 2018.

[20] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering (ICSE)*, 2009.

[21] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[23] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[24] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. Technical report, January 2017.

[25] Peter Goodman. Shin GRR: Make Fuzzing Fast Again. https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again/.

[26] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, 2013.

[27] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[28] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. Enhancing memory error detection for large-scale applications and fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[29] Aki Helin. A general-purpose fuzzer. https://gitlab.com/akihe/radamsa. Accessed: 2018-08-07.

[30] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[31] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[33] David Molnar, Xue Cong Li, and David Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, 2009.

[34] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[35] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, February 2017.

[36] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan M Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[37] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *USENIX Security Symposium*, 2017.

[38] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[39] Dmitry Vyukov. gofuzz. https://go-talks.appspot.com/github.com/dvyukov/go-fuzz/slides/go-fuzz.slide#17.

[40] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[41] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[42] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[43] Michał Zalewski. afl-fuzz: making up grammar with a dictionary in hand. https://lcamtuf.blogspot.de/2015/01/afl-fuzz-making-up-grammar-with.html. Accessed:2018-08-07.

[44] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2018-08-07.