Driller: Augmenting Fuzzing Through Selective Symbolic Execution

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang,
Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna
UC Santa Barbara
{*stephens,jmg,salls,dutcher,fish,jacopo,yans,chris,vigna*}*@cs.ucsb.edu*

***Abstract*—Memory corruption vulnerabilities are an ever-present risk in software, which attackers can exploit to obtain unauthorized access to confidential information. As products with access to sensitive data are becoming more prevalent, the number of potentially exploitable systems is also increasing, resulting in a greater need for automated software vetting tools. DARPA recently funded a competition, with millions of dollars in prize money, to further research focusing on automated vulnerability finding and patching, showing the importance of research in this area. Current techniques for finding potential bugs include static, dynamic, and concolic analysis systems, which each having their own advantages and disadvantages. A common limitation of systems designed to create inputs which trigger vulnerabilities is that they only find shallow bugs and struggle to exercise deeper paths in executables.**

We present Driller, a hybrid vulnerability excavation tool which leverages fuzzing and selective concolic execution in a complementary manner, to find deeper bugs. Inexpensive fuzzing is used to exercise *compartments* of an application, while concolic execution is used to generate inputs which satisfy the complex checks separating the compartments. By combining the strengths of the two techniques, we mitigate their weaknesses, avoiding the path explosion inherent in concolic analysis and the incompleteness of fuzzing. Driller uses selective concolic execution to explore only the paths deemed interesting by the fuzzer and to generate inputs for conditions that the fuzzer cannot satisfy. We evaluate Driller on 126 applications released in the qualifying event of the DARPA Cyber Grand Challenge and show its efficacy by identifying the same number of vulnerabilities, in the same time, as the top-scoring team of the qualifying event.

I. Introduction

Despite efforts to increase the resilience of software against security flaws, vulnerabilities in software are still commonplace. In fact, in recent years, the occurrence of security vulnerabilities has increased to an all-time high [28]. Furthermore, despite the introduction of memory corruption and execution redirection mitigation techniques, such software flaws account for over a third of all vulnerabilities discovered in the last year [14].

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23368

Whereas such vulnerabilities used to be exploited by independent hackers who wanted to push the limits of security and expose ineffective protections, the modern world has moved to nation states and cybercriminals using such vulnerabilities for strategic advantage or profit. Furthermore, with the rise of the *Internet of Things*, the number of devices that run potentially vulnerable software has skyrocketed, and vulnerabilities are increasingly being discovered in the software running these devices [29].

While many vulnerabilities are discovered by hand, manual analysis is not a scalable method for vulnerability assessment. To keep up with the amount of software that must be vetted for vulnerabilities, an automated approach is required. In fact, DARPA has recently lent its support to this goal by sponsoring two efforts: VET, a program on developing techniques for the analysis of binary firmware, and the Cyber Grand Challenge (CGC), in which participants design and deploy automated vulnerability scanning engines that will compete against each other by exploiting binary software. DARPA has funded both VET and the Cyber Grand Challenge with millions of dollars in research funding and prize money, demonstrating the strong interest in developing a viable approach to automated binary analysis.

Naturally, security researchers have been actively designing automated vulnerability analysis systems. Many approaches exist, falling into three main categories: static, dynamic, and concolic analysis systems. These approaches have different advantages and disadvantages. Static analysis systems can provide provable guarantees – that is, a static analysis system can show, with certainty, that a given piece of binary code is secure. However, such systems have two fundamental drawbacks: they are imprecise, resulting in a large amount of false positives, and they cannot provide "actionable input" (i.e., an example of a specific input that can trigger a detected vulnerability). Dynamic analysis systems, such as "fuzzers", monitor the native execution of an application to identify flaws. When flaws are detected, these systems can provide actionable inputs to trigger them. However, these systems suffer from the need for "input test cases" to drive execution. Without an exhaustive set of test cases, which requires considerable manual effort to generate, the usability of such systems is limited. Finally, concolic execution engines utilize program interpretation and constraint solving techniques to generate inputs to explore the state space of the binary, in an attempt to reach and trigger vulnerabilities. However, because such systems are able to trigger a large number of paths in the binary (i.e., for a conditional branch, they often create an input that

causes the branch to be taken and another that does not), they succumb to "path explosion", greatly limiting their scalability.

Because of these drawbacks, most bug-triggering input produced by modern automated analysis systems represents "shallow" bugs in the software. In the case of fuzzers, this is because fuzzers randomly generate new inputs to an application and they likely fail to successfully pass through input-processing code. Concolic execution engines, on the other hand, are often able to recreate properly formatted input to pass through input processing code, but tend to succumb to path explosion, limiting the "depth" of code that they can analyze. Thus, flaws that lie in the deeper logic of an application tend to be missed by these tools, and are usually discovered through manual analysis by human experts [3], [9], [13].

The difference between the types of bugs that can be found by fuzzing and concolic execution can also be viewed in terms of the way in which an application processes user input. We propose two different categories of user input: *general* input, which has a wide range of valid values (e.g., the name of a user) and *specific* input, which has a limited set of valid values (e.g., the hash of the aforementioned name). An application's checks for particular values of specific input effectively split an application into *compartments*, separated by such checks. Fuzzing is proficient at exploring possible values of general input, *within* a compartment, but struggles to identify the precise values needed to satisfy checks on specific input and drive execution flow *between* compartments. On the other hand, selective concolic execution is proficient at determining the values that such specific checks require and, if the path explosion problem were solved, can push execution between compartments.

For example, consider an application that processes commands from the user: the application reads a command name, compares it against a list of commands, and passes user-supplied parameters to the appropriate command handler. In this case, the complex check would be the comparison of the command name: a fuzzer randomly mutating input would have a very small chance of sending the correct input. On the other hand, a concolic execution engine would be well-suited for recovering the correct command name, but might suffer a path explosion in the parameter-processing code. Once the correct command name is determined, a fuzzer is better-suited for exploring the different command parameters that could be sent, without encountering a path explosion.

We realized that this observation can be used to combine multiple analysis techniques, leveraging their strengths while mitigating their weaknesses. For example, a fuzzer can be used to explore the initial compartment of an application and, when it is unable to go further, a concolic execution engine can be leveraged to guide it to the next compartment. Once there, the fuzzer can take over again, exploring the possible inputs that can be provided to the new compartment. When the fuzzer stops making progress again, the concolic execution engine can resume and direct the analysis to the next compartment, and so on. By doing this repeatedly, execution is driven deeper and deeper into the program, limiting the path explosion inherent to concolic execution and ameliorating the incompleteness of dynamic analysis.

Guided by this intuition, we created a system, *Driller*, that is a novel vulnerability excavation system combining a genetic input-mutating fuzzer with a selective concolic execution engine to identify deep bugs in binaries. Combining these two techniques allows Driller to function in a scalable way and bypass the requirement of input test cases. In this paper, we will describe the design and implementation of Driller and evaluate its performance on 126 applications released as part of the qualifying event of the DARPA Cyber Grand Challenge.

Driller is not the first work to combine different types of analyses. However, existing techniques either support very specific types of vulnerabilities (while Driller currently detects any vulnerability that can lead to a program crash) [21], [25], do not take full advantage of the capabilities offered by dynamic analysis (and, specifically, fuzzing) [19], or are affected by the path explosion problem [4], [8], [10], [20]. We show that Driller identifies more vulnerabilities in these binaries than can be recovered separately by either fuzzing or concolic execution, and demonstrate the efficacy of our approach by discovering the same number of vulnerabilities, within the same amount of time, on the same dataset, as the winning team of the Cyber Grand Challenge qualifying event. Furthermore, we perform additional evaluations to show that this would not be possible without Driller's contribution (i.e., using traditional fuzzing or symbolic execution approaches).

In summary, this paper makes the following contributions:

- We propose a new method to improve the effectiveness of fuzzing by leveraging selective concolic execution to reach deeper program code, while improving the scalability of concolic execution by using fuzzing to alleviate path explosion.
- We designed and implemented a tool, Driller, to demonstrate this approach.
- We demonstrate the effectiveness of Driller by identifying the same number of vulnerabilities, on the same dataset, as the winning team of the Cyber Grand Challenge qualifying event.

II. Related Work

Driller is a guided whitebox fuzzer which builds on top of state-of-the-art fuzzing techniques, adding concolic execution to achieve effective vulnerability excavation. As some other existing vulnerability excavation tools also combine multiple techniques, we will use this section to distinguish Driller from other solutions which draw on related techniques.

A. Guided Fuzzing

Fuzzing was originally introduced as one of several tools to test UNIX utilities [23]. Since then, it has been extensively used for the black-box security testing of applications. However, fuzzing suffers from a lack of guidance – new inputs are generated based on random mutations of prior inputs, with no control over which paths in the application should be targeted.

The concept of guided fuzzing arose to better direct fuzzers toward specific classes of vulnerabilities. For example, many studies have attempted to improve fuzzing by selectively choosing optimal test cases, honing in on interesting regions of code contained in the target binary [21], [25]. Specifically,

Dowser [21] uses static analysis to first identify regions of code that are likely to lead to a vulnerability involving a buffer overflow. To analyze this code, Dowser applies taint-tracking to available test cases to determine which input bytes are processed by these code regions and symbolically explores the region of code with only these bytes being symbolic. Unfortunately, Dowser has two drawbacks: it requires test cases to reach the region of code containing the memory corruption vulnerability, and it only supports buffer overflow vulnerabilities. Unlike Dowser, Driller supports arbitrary vulnerability specifications (though the current implementation focuses on vulnerabilities that lead to a crash) and does not require input test cases. Additionally, Dowser still suffers from the path explosion problem of symbolic execution, while Driller mitigates this problem through its use of fuzzing.

Similar to Dowser, BuzzFuzz [17] applies taint-tracking to sample input test cases to discover which input bytes are processed by 'attack-points' defined by the auditor, most often system call arguments and library code. Unlike BuzzFuzz, Driller does not rely on input test cases that reach vulnerable code, nor does it rely on auditor defined 'attack-points'.

In another attempt to improve the state of fuzzing, Flayer [15] allows an auditor to skip complex checks in the target application at-will. This allows the auditor to fuzz logic deeper within the application without crafting inputs which conform to the format required by the target, at the cost of time spent investigating the validity of crashing inputs found. Similarly, Taintscope uses a checksum detection algorithm to remove checksum code from applications, effectively "patching out" branch predicates which are difficult to satisfy with a mutational approach [30]. This enables the fuzzer to handle specific classes of difficult constraints. Both these approaches, however, either require a substantial amount of human guidance in Flayer's case, or manual effort to determine false positives during crash triaging. Driller does not modify any code of the target application, meaning crashes discovered do not require an in-depth investigation, additionally Driller does not require human intervention, as it attempts to discover well-formed inputs using its concolic execution backend.

Another approach is Hybrid Fuzz Testing, in which limited symbolic exploration is utilized to find "frontier nodes" [26]. Fuzzing is then employed to execute the program with random inputs, which are preconstrained to follow the paths leading to a frontier node. This method is useful for ensuring that the fuzzed inputs take different paths early in the execution of the binary, but it does not handle complex checks, deeper in the program, which separate compartments. Additionally, the path explosion problem effectively prevents the symbolic exploration from solving more than just the shallow checks in the binary.

B. Whitebox Fuzzing

Other systems attempt to blend fuzzing with symbolic execution to gain maximal code coverage [6], [7], [19], [20]. These approaches tend to augment fuzzing by symbolically executing input produced by a fuzzing engine, collecting symbolic constraints placed on that input, and negating these constraints to generate inputs that will take other paths. However, these tools lack Driller's key insight, that symbolic execution is best used to recover input for driving code execution between application compartments. Without this insight, the unique capabilities of symbolic execution are wasted on creating divergent paths *within* compartments. These tools are, in essence, symbolic execution engines acting in a *serialized* manner, one path at a time, and as such, they are deeply affected by the path explosion problem.

While Driller is similar in a number of implementation details, we propose that we can offload the majority of unique path discovery to an instrumented fuzzing engine. We limit our costly symbolic execution invocations to satisfy conditions that will allow us to enter additional compartments for fuzzing. Since we only use symbolic execution for generating the basic block transitions that the fuzzer has not been able to generate itself, the symbolic execution engine handles a manageable number of inputs. Conversely, the aforementioned tools repetitively negate constraints using concolic execution, slowly analyzing an exponentially increasing number of transitions, most of which can be analyzed more efficiently by a fuzzer.

C. Concolic Execution

With the continuing increase of computing power in recent years, *concolic execution* (also known as *dynamic symbolic execution*) has risen in popularity. Introduced with EXE [5], refined with KLEE [4], and applied to binary code with Mayhem [8] and S2E [10], concolic execution engines interpret an application, model user input using symbolic variables, track constraints introduced by conditional jumps, and use constraint solvers to create inputs to drive applications down specific paths. While these systems are powerful, they suffer from a fundamental problem: if a conditional branch depends on symbolic values, it is often possible to satisfy *both* the taken and non-taken condition. Thus, the state has to *fork* and both paths must be explored. This quickly leads to the well-known path explosion problem, which is the primary inhibitor of concolic execution techniques.

Various approaches have been attempted to mitigate the path explosion problem. Veritesting [1] proposed an advanced path merging technique to reduce the number of paths being executed, Firmalice [29] performs extensive static analysis and limits symbolic execution to small slices of code, and under-constrained symbolic execution exchanges precision for scalability [16], [27]. However, these techniques either fail to mitigate the path explosion problem (Veritesting delays the explosion, but such explosion still eventually occurs) or produce inputs that are not directly actionable (for example, the slicing done by Firmalice produces inputs that satisfy the constraints of a particular slice, but no input is provided to *reach* the code in the first place).

Driller attempts to mitigate this by offloading most of the path exploration task to its fuzzing engine, using concolic execution only to satisfy complex checks in the application that guard the transition between compartments.

III. Driller Overview

A core intuition behind the design of Driller is that applications process two different classes of user input: *general* input, representing a wide range of values that can be valid, and *specific* input, representing input that must

take on one of a select few possible values. Conceptually, an application's checks on the latter type of input split the application into *compartments*. Execution flow moves between compartments through checks against specific input, while, within a compartment, the application processes general input. This concept is explored in more depth in Section VI-G in the context of an actual binary in our experimental dataset.

Driller functions by combining the speed of fuzzing with the input reasoning ability of concolic execution. This allows Driller to quickly explore portions of binaries that do not impose complex requirements on user input while also being able to handle, without the scalability issues of pure concolic execution, complex checks on specific input. In this paper, we define "complex" checks as those checks that are too specific to be satisfied by input from an input-mutating fuzzer.

Driller is composed of multiple components. Here, we will summarize these components and provide a high-level example of Driller's operation. In the rest of the paper, we will describe these components in depth.

Input test cases. Driller can operate without input test cases. However, the presence of such test cases can speed up the initial fuzzing step by pre-guiding the fuzzer toward certain compartments.

Fuzzing. When Driller is invoked, it begins by launching its fuzzing engine. The fuzzing engine explores the first compartment of the application until it reaches the first complex check on specific input. At this point, the fuzzing engine gets "stuck" and is unable to identify inputs to search new paths in the program.

Concolic execution. When the fuzzing engine gets stuck, Driller invokes its selective concolic execution component. This component analyzes the application, pre-constraining the user input with the unique inputs discovered by the prior fuzzing step to prevent a path explosion. After tracing the inputs discovered by the fuzzer, the concolic execution component utilizes its constraint-solving engine to identify inputs that would force execution down previously unexplored paths. If the fuzzing engine covered the previous compartments before getting stuck, these paths represent execution flows into new compartments.

Repeat. Once the concolic execution component identifies new inputs, they are passed back to the fuzzing component, which continues mutation on these inputs to fuzz the new compartments. Driller continues to cycle between fuzzing and concolic execution until a crashing input is discovered for the application.

A. Example

To elucidate the concept behind Driller, we provide an example in Listing 1. In this example, the application parses a configuration file, containing a magic number, received over an input stream. If the received data contains syntax errors or an incorrect magic number, the program exits. Otherwise, control flow switches based on input between a number of new compartments, some of which contain memory corruption flaws.

Driller begins its operation by invoking its fuzzing engine and fuzzing the first compartment of the application. These fuzzed nodes are shown shaded in a control-flow graph of the program in Figure 1. This fuzzing step explores the first compartment and gets stuck on the first complex check – the comparison with the magic number. Then, Driller executes the concolic execution engine to identify inputs that will drive execution past the check, into other program compartments. The extra transitions discovered by the concolic execution component, for this example, are shown in Figure 2.

After this, Driller enters its fuzzing stage again, fuzzing the second compartment (the initialization code and the check against keys in the configuration file). The coverage of the second fuzzing stage is shown in Figure 3. As shown, the fuzzer cannot find any arms of the key switch besides the default. When this second fuzzing invocation gets stuck, Driller leverages its concolic execution engine to discover the "`crashstring`" and "`set_option`" inputs, shown in Figure 4. The former leads directly to the bug in the binary.

It is important to note that while neither symbolic execution nor fuzzing by themselves could find this bug, Driller can. There are several areas in this example where Driller's hybrid approach is needed. The parsing routines and initialization code have a great amount of complicated control flow reasoning about highly stateful data, which would lead to path explosion, slowing down symbolic execution to the point of uselessness. Additionally, and as noted before, the magic number check foils traditional fuzzing approaches by requiring highly specific input, too small to be reasonably found within its search space. Other common programming techniques that hinder fuzzing approaches include the use of hash functions to validate input. For this reason, a composition of concolic execution and fuzzing has the potential of achieving better results.

```
1  int main(void) {
2    config_t *config = read_config();
3    if (config == NULL) {
4      puts("Configuration syntax error");
5      return 1;
6    }
7    if (config->magic != MAGIC_NUMBER) {
8      puts("Bad magic number");
9      return 2;
10   }
11   initialize(config);
12
13   char *directive = config->directives[0];
14   if (!strcmp(directive, "crashstring")) {
15     program_bug();
16   }
17   else if (!strcmp(directive, "set_option")) {
18     set_option(config->directives[1]);
19   }
20   else {
21     default();
22   }
23
24   return 0;
25 }
```

Listing 1. An example requiring fuzzing and concolic execution to work together.

IV. Fuzzing

Fuzzing is a technique that executes an application with a wide set of inputs, checking if these inputs cause the application to crash. To retain speed of execution, fuzzers are minimally invasive – they perform minimal instrumentation on the underlying application and mostly monitor it from the outside.

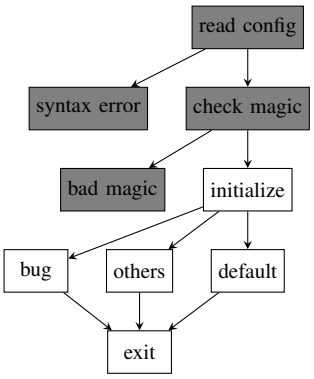


Fig. 1. The nodes initially found by the fuzzer.

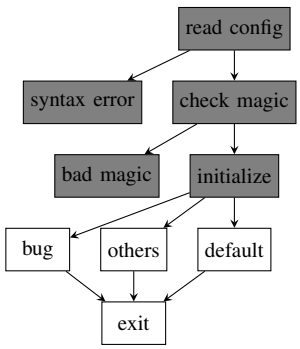


Fig. 2. The nodes found by the first invocation of concolic execution.

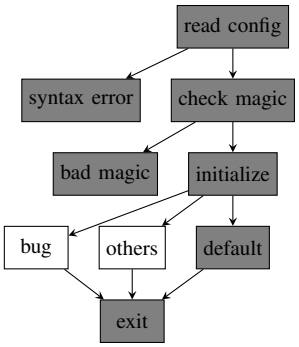


Fig. 3. The nodes found by the fuzzer, supplemented with the result of the first Driller run.

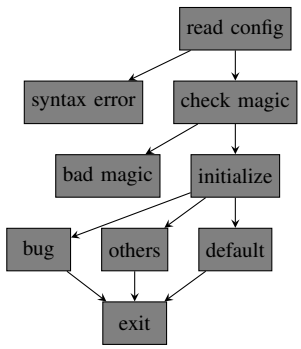


Fig. 4. The nodes found by the second invocation of concolic execution.

Recent years have seen many improvements to fuzzing engines. In this section, we will detail improvements that are relevant to Driller's performance.

To implement Driller, we leveraged a popular off-the-shelf fuzzer, American Fuzzy Lop (AFL) [31]. Our improvements mostly deal with integrating the fuzzer with our concolic execution engine. No changes to the logic of AFL were made. AFL relies on instrumentation to make informed decisions on which paths are interesting. This instrumentation can be either introduced at compile-time or via a modified QEMU [2], we opted for a QEMU-backend to remove reliance on source code availability. While we discuss important features of Driller's fuzzer-component, AFL, in this section, we do not claim credit for their invention or implementation.

A. Fuzzer Features

A modern fuzzer implements many features to better identify crashing inputs. In this section, we will list and describe the most important AFL features, mentioning how they are used by Driller.

Genetic fuzzing. AFL carries out input generation through a genetic algorithm, mutating inputs according to genetics-inspired rules (transcription, insertion, etc.) and ranking them by a fitness function. For AFL, the fitness function is based on *unique* code coverage – that is, triggering an execution path that is different than the paths triggered by other inputs.

State transition tracking. AFL tracks the union of control flow transitions that it has seen from its inputs, as tuples of the source and destination basic blocks. Inputs are prioritized for "breeding" in the genetic algorithm based on their discovery of new control flow transitions, meaning that inputs that cause the application to execute in a different way get priority in the generation of future inputs.

Loop "bucketization". Handling loops is a complicated problem for fuzzing engines and concolic execution engines alike. To help reduce the size of the path space for loops, the following heuristic is performed. When AFL detects that a path contains iterations of a loop, a secondary calculation is triggered to determine whether that path should be eligible for breeding. AFL determines the number of loop iterations that were executed and compares it against previous inputs that caused a path to go through the same loop. These paths are all placed into "buckets" by the logarithm of their loop iteration count (i.e., 1, 2, 4, 8, and so on). One path from each bucket is considered for breeding in the genetic algorithm. This way, only $\log(N)$ paths must be considered for each loop as opposed to the naive approach of N paths.

Derandomization. Program randomization interferes with a genetic fuzzer's evaluation of inputs – an input that produces interesting paths under a given random seed might not do so under another. We pre-set AFL's QEMU backend to a specific random seed to ensure consistent execution. Later, when a crashing input is discovered, we use our concolic execution engine to recover any "challenge-response" behavior or vulnerabilities that rely on leaking randomness. For example, a "challenge-response" process in a binary echoes random data to the user and expects the same data echoed back to it. Without removing randomization, the fuzzing component would likely fail this check every time and explore very few paths. If the randomness is instead constant, the program accepts the same input each time, leaving the fuzzer (or the concolic execution component) free to find this one value and subsequently explore further. After a crash is found, the randomness can instead be modeled symbolically, as described in section V-D4, and the crashing input can be patched accordingly.

These features allow AFL to rapidly discover unique paths through an application, performing the brunt of the path discovery work within a given compartment of the application. However, the limitations of fuzzing are well-known.

B. Fuzzer Limitations

Because fuzzers randomly mutate input, and genetic fuzzers, in turn, mutate input that has, in the past, generated unique paths through a binary, they are able to quickly discover different paths that process "general" input (i.e., input that has many different values that can trigger meaningful program behavior). However, the generation of "specific" input to pass complex checks in the application (i.e., checks that require inputs with one of very few specific values) is very challenging for fuzzers.

```
1 int main(void)
2 {
3     int x;
4     read(0, &x, sizeof(x));
5
6     if (x == 0x0123ABCD)
7         vulnerable();
8 }
```

Listing 2. A difficult program to fuzz.

The application in Listing 2 reads a value from the user and compares it against a specific value. If the correct value is provided, the application will crash. However, due to the nature of fuzzing, it is extremely unlikely that a fuzzer will ever satisfy the predicate. For a non-instrumented fuzzer (i.e., one that chooses random values for the input), the likelihood that the fuzzer will discover the bug is the infinitesimal 1 out of 2^{32}. For an instrumented fuzzer, the control flow layout of this binary will result in a single path being discovered. Without the ability to prioritize new paths (as there are none), an instrumented fuzzer will be reduced to applying random mutations on the existing paths which is, in essence, the same as the non-instrumented case, with the same infinitesimally small chance of success.

C. Transition to Concolic Execution

Driller aims to complement the fundamental weakness of fuzzing, determining specific user input required to pass complex checks, by leveraging the strength of concolic execution. When the fuzzing component has gone through a predetermined amount (proportional to the input length) of mutations without identifying new state transitions, we consider it "stuck". Driller then retrieves the inputs that the fuzzer has deemed "interesting" in the current compartment and invokes the concolic execution engine on them.

The fuzzer identifies inputs as interesting if one of two conditions holds:

1) The path that the input causes the application to take was the first to trigger some state transition.
2) The path that the input causes the application to take was the first to be placed into a unique "loop bucket".

These conditions keep the number of inputs that are handed to the concolic execution component down to a reasonable number, while retaining a high chance of passing along inputs that the concolic execution can mutate to reach the next compartment in the application.

V. Selective Concolic Execution

When Driller determines that the fuzzer is unable to find additional state transitions, the concolic execution engine is invoked. The insight behind Driller's use of concolic execution is as follows: one of the main causes of fuzzers failing to find new state transitions in a program is the inability of fuzzers to generate specific input to satisfy complex checks in the code. The concolic execution engine is used to leverage a symbolic solver to *mutate* existing inputs that reach but fail to satisfy complex checks into new inputs that reach and *satisfy* such checks.

When Driller invokes the concolic execution engine, it passes all of the "interesting" inputs (as defined in Section IV-C) that were identified by the fuzzing engine. Each input is traced, symbolically, to identify state transitions that the fuzzing engine was unable to satisfy. When such a transition is identified, the concolic execution engine produces input that would drive execution through this state transition.

After the concolic execution engine finishes processing the provided inputs, its results are fed back into the fuzzing engine's queue and control is passed back to the fuzzing engine, so that it can quickly explore the newly found compartments of the application.

The remainder of this section will describe Driller's implementation of concolic execution and the specific adaptations that we made for Driller's problem domain.

A. Concolic Execution

We leveraged angr [29], a recently open-sourced symbolic execution engine, for Driller's concolic execution engine. The engine is based on the model popularized and refined by Mayhem and S2E [8], [10]. First the engine translates binary code into Valgrind's VEX [24] intermediate representation, which is interpreted to determine the effects of program code on a *symbolic state*. This symbolic state uses *symbolic variables* to represent input that can come from the user or other data that is not constant, such as data from the environment. A symbolic variable is a variable (such as X) that can yield a number of possible concrete solutions (such as the number 5). Other values, such as constants hardcoded in the program, are modeled as concrete values. As the execution progresses, *symbolic constraints* are added to these variables. A constraint is a limiting statement on the potential solutions of the symbolic value (for example, $X < 100$). A concrete solution is any value of X that will satisfy these constraints.

The analysis engine tracks all concrete and symbolic values in memory and registers (the aforementioned symbolic state) throughout execution. At any point in the program that the engine reaches, a constraint resolution can be performed to determine a possible input that satisfies the constraints on all symbolic variables in the state. Such an input, when passed to a normal execution of the application, would drive the application to that point. The advantage of concolic execution is that it can explore and find inputs for any path that the constraint solver can satisfy. This makes it useful for identifying solutions to complex comparisons (up to and including certain hash functions) that a fuzzer would be unlikely to ever brute force.

Driller's symbolic memory model can store both concrete and symbolic values. It uses an index-based memory model in which read addresses may be symbolic, but write address are always concretized. This approach, popularized by Mayhem, is an important optimization to keep the analysis feasible: if both read and write addresses were symbolic, a repeated read and write using the same symbolic index would result in a quadratic increase in symbolic constraints or, depending on the implementation details of the symbolic execution engine, the complexity of the stored symbolic expressions. Thus, symbolic write addresses are always concretized to a single valid solution. Under certain conditions, as proposed by literature in the field, symbolic values are concretized to a single potential solution [8].

The symbolic memory optimizations increase the scalability of the concolic execution engine, but can result in an incomplete state space, where fewer solutions are possible. Unfortunately, this is a trade-off that must be made to make analysis of real-world binaries realistic.

B. Example

Concolic execution is good at solving different problems than fuzzing. Recall the example demonstrating the drawback of fuzzing, from Section IV-B, reproduced in Listing 3.. Because of the exactness of the input required to pass the check guarding the call to the `vulnerable` function, fuzzing is unable to explore that piece of code in a reasonable time frame.

```
1 int main(void)
2 {
3     int x;
4     read(0, &x, sizeof(x));
5
6     if (x == 0x0123ABCD)
7         vulnerable();
8 }
```

Listing 3. A program that yields to concolic execution.

However, a concolic execution engine will be able to easily satisfy this check and trigger the `vulnerable` function. For this example, concolic execution only needs to explore a small number of paths to find one which reaches the bug in this example, but for bigger binaries and real-world examples, there will be far too many paths to explore in the same manner.

C. Limitations

The traditional approach to concolic execution involves beginning concolic execution from the beginning of a program and exploring the path state with the symbolic execution engine to find as many bugs as possible. However, this approach suffers from two major limitations.

First, concolic execution is slow. This is caused by the need to interpret application code (as opposed to natively executing it, as with a fuzzer) and by the overhead involved in the constraint solving step. Specifically, the latter operation involves the solution of an NP-complete problem, making the generation of potential inputs (and the determination of which conditional jumps are feasible) time-consuming.

Worse, symbolic execution suffers from the state explosion problem. The number of paths grows exponentially as the concolic execution engine explores the program, and it quickly becomes infeasible to explore more than a tiny fraction of the paths. Consider the example in Listing 4. In this program, the `vulnerable()` is triggered when the user enters exactly 25 `B` characters, but this is a condition difficult to express in a symbolic execution framework. Symbolic execution of this program will cause a huge state explosion as the simulated CPU steps down the recursive calls into the `check()` function. Each execution of the ternary conditional comparing a character to the literal `B` splits every simulated state into two, eventually resulting in 2^{100} possible states, which is an infeasible amount to process.

A genetic fuzzer that selects inputs based on state transitions, on the other hand, does not reason about the whole state-space of a program, but only on the state transitions triggered by inputs. That is, it will focus chiefly on the number of times, for example, the check on line 5 succeeds. That is, regardless of *where* the `B` characters are in the input, states will be judged based on the *number* of them in the input, avoiding the path explosion problem.

While progress has been made toward reducing this problem with intelligent state merging [1], the general problem remains.

```
1  int check(char *x, int depth) {
2    if (depth >= 100) {
3      return 0;
4    } else {
5      int count = (*x == 'B') ? 1 : 0;
6      count += check(x+1, depth+1);
7      return count;
8    }
9  }
10
11 int main(void) {
12   char x[100];
13   read(0, x, 100);
14
15   if (check(x, 0) == 25)
16     vulnerable();
17 }
```

Listing 4. A program that causes a path explosion under concolic execution.

D. Concolic Execution in Driller

In most cases, fuzzing can adequately explore a large portion of paths on its own, simply by finding them with random bit flips and other mutation strategies. By utilizing native execution, it will outperform concolic execution in most cases where it can randomly trigger the paths. Thus, most of the work is offloaded from the concolic execution engine to the fuzzer, which will find many paths quickly, letting the concolic engine just work on solving the harder constraints.

When fuzzing is unable to discover inputs that result in new execution paths, the concolic execution engine is invoked. It traces the paths discovered by the fuzzing, identifies inputs that diverge into new program components, and performs limited symbolic exploration. Additionally, when a crashing input is found by the fuzzing component, the concolic execution engine "re-randomizes" it to recover the parts of a crashing input that are dependent on randomness and other environmental factors.

1) Pre-constrained Tracing: Driller uses concolic execution to trace the interesting paths from the fuzzer and generate new inputs. A key factor in the effectiveness of this approach is that it allows Driller to avoid the path explosion inherent in concolic exploration, because only the path representing the application's processing of that input is analyzed.

When traces are passed from the fuzzer to the symbolic execution, the goal is to discover new transitions that fuzzing had not previously found. Driller's concolic execution engine traces the input, following the same path that was taken by the fuzzer. When Driller comes upon a conditional control flow transfer, it checks if *inverting* that condition would result in the discovery of a new state transition. If it will, Driller produces an example input that will drive execution through the new state transition instead of the original control flow.

By doing this Driller's concolic execution engine guides the fuzzing engine to new compartments of the application. After producing the input, Driller continues following the matching path to find additional new state transitions.

2) Input Preconstraining: Driller uses *preconstraining* to ensure that the results of the concolic execution engine are identical to those in the native execution while maintaining the ability to discover new state transitions. In preconstrained execution, each byte of input is constrained to match each actual byte that was output by the fuzzer, e.g., `/dev/stdin[0] == 'A'`. When new possible basic block transitions are discovered, the preconstraining is briefly removed, allowing Driller to solve for an input that would deviate into that state transition. Preconstraining is necessary to generate identical traces in the symbolic execution engine and make the limited concolic exploration feasible.

```
1  int check(char *x, int depth) {
2    if (depth >= 100) {
3      return 0;
4    } else {
5      int count = (*x == 'B') ? 1 : 0;
6      count += check(x+1, depth+1);
7      return count;
8    }
9  }
10
11 int main(void) {
12   char x[100];
13   int magic;
14   read(0, x, 100);
15   read(0, &magic, 4);
16
17   if (check(x, 0) == 25)
18     if (magic == 0x42d614f8)
19       vulnerable();
20 }
```

Listing 5. An application showcasing the need for pre-constraining of symbolic input.

To demonstrate how input preconstraining works in Driller, we use the example in Listing 5, which is similar to the example from Section V-C with the addition that, to reach the vulnerable function, we must provide a magic number (0x42d614f8) at line 18. After fuzzing the input, Driller eventually recognizes that it is not discovering any new state transitions, since the fuzzer alone cannot guess the correct value. When concolic execution is invoked to trace an input, Driller first constrains all of the bytes in the symbolic input to match those of the traced input. As the program is symbolically executed, there is only one possibility for each branch, so exactly one path is followed. This prevents the path explosion that was described in Section V-C. When execution reaches line 18, however, Driller recognizes that there is an alternate state transition that has never been taken during fuzzing. Driller then removes the preconstraints that were added at the beginning of the execution not including the predicates placed by symbolically executing the program with the traced input. The bytes in the character array `x` are partially constrained by the path, and the value of `magic` is constrained by the equality check `if (magic == 0x42d614f8)`. The concolic execution engine thus creates an input that contains 25 instances of `B` and a `magic` value of 0x42d614f8. This passes the check in line 18 and reaches the vulnerable function.

3) Limited Symbolic Exploration: In an attempt to reduce the number of expensive concolic engine invocations we also introduce a symbolic exploration stub to discover more state transitions lying directly after a newly discovered state transition. This symbolic exploration stub explores the surrounding area of the state transition until a configurable number of basic blocks has been traversed by the explorer. Once this number of blocks has been discovered, Driller concretizes inputs for all paths discovered by the explorer. We reason that doing this prevents the fuzzer from getting "stuck" quickly after being provided with a Driller-generated input. In a number of cases, Driller generates a new input that gets only partway through a multi-part complex check and must immediately be retraced to allow the fuzzer to proceed deeper into the binary. The symbolic exploration stub is a small optimization which allows Driller to find further state transitions, before they are requested, without having to retrace its steps.

4) Re-randomization: Random values introduced during a program run can disrupt fuzzing attempts as described earlier. Listing 6 displays a small program which challenges the user to reflect back a random input. This makes fuzzing unstable because we can never know the concrete value of `challenge` without monitoring the program output.

```
1  int main(void) {
2      int challenge;
3      int response;
4
5      challenge = random();
6
7      write(1, &challenge, sizeof(challenge));
8      read(0, &response, sizeof(response));
9      if (challenge == response)
10         abort();
11
12 }
```

Listing 6. A program which requires re-introducing randomness.

Once a vulnerability is discovered, we use symbolic execution to trace crashing inputs and recover input bytes that need to satisfy dynamic checks posed by the target binary (such as the challenge-response in the example of Listing 6). By inspecting the symbolic state at crash time and finding the relationships between the application's output and the crashing input, Driller can determine the application's challenge-response protocol. In this example, we can see that the symbolic bytes introduced by the call to `read` are constrained to being equal to the bytes written out by the call to `write`. After determining these relationships, we can generate an exploit specification that handles randomness as it occurs in a real environment.

VI. Evaluation

To determine the effectiveness of our approach, we performed an evaluation on a large dataset of binaries. The goal of our evaluation is to show two things: first, Driller considerably expands the code coverage achieved by an unaided fuzzer, and, second, this increased coverage leads to an increased number of discovered vulnerabilities.

A. Dataset

We evaluated Driller on applications from the qualifying event of the DARPA Cyber Grand Challenge (CGC) [11], a

competition designed to "test the abilities of a new generation of fully automated cyber defense systems" [11]. During the event, competitors had 24 hours to autonomously find memory corruption vulnerabilities and demonstrate proof by providing an input specification that, when processed by the application in question, causes a crash. There are 131 services in the CGC Qualifying Event dataset, but 5 of these involve communication between multiple binaries. As such functionality is out of scope for this paper, we only consider the 126 single-binary applications, leaving multi-binary applications to future work.

These 126 applications contain a wide range of obstacles that make binary analysis difficult, such as complex protocols and large input spaces. They are specifically created to stress the capabilities of program analysis techniques, and are not simply toy applications for hacking entertainment (unlike what is generally seen at Capture The Flag hacking competitions [22]). The variety and depth of these binaries allow for extensive testing of advanced vulnerability excavation systems, such as Driller. Furthermore, the results of the top competitors are available online, providing a litmus test for checking the performance of analysis systems against verified results.

B. Experiment Setup

We ran our experiments on a computer cluster of modern AMD64 processors. Each binary had four dedicated fuzzer nodes and, when the fuzzer requires concolic execution assistance, it sent jobs to a pool of 64 concolic execution nodes, shared among all binaries. Due to constraints on the available memory, we limited each concolic execution job to 4 gigabytes of RAM. In all of our tests, we analyze a single binary for at most 24 hours, which is the same amount of time that was given to the teams for the CGC qualifying event. We analyzed each binary until either a crash was found or the 24 hours had passed.

All crashes were collected and replayed using the challenge binary testing tools to verify that the reported crashes were repeatable in the actual CGC environment. Thus, these results are *real*, verified, and comparable to the actual results from the competition.

C. Experiments

We ran a total of three experiments in our evaluation. First, to evaluate Driller against the baseline performance of existing techniques, we attempted vulnerability excavation with a pure symbolic execution engine and a pure fuzzer. Then, we evaluated Driller on the same dataset.

The experiments were set up as follows:

Basic fuzzing. In this test, each binary was assigned 4 cores for fuzzing by AFL, but the concolic execution nodes were deactivated. The fuzzer had no assistance when it was unable to discover new paths. Note that changes were made to AFL's QEMU backend to improve performance on CGC binaries, however, as mentioned previously no core changes to AFL's logic were made.

Symbolic execution. We used an existing symbolic execution engine, based heavily on the ideas proposed by Mayhem [8], for the concolic execution test. To ensure a fair test against the state of the art, advanced state merging techniques were used to help limit the effects of state explosion, as proposed in Veritesting [1].
We analyze each binary by symbolically exploring the state space, starting from the entry point, checking for memory corruption. When a state explosion did occur, we used heuristics to prioritize paths that explored deeper into the application to maximize code coverage.

Driller. When testing Driller, each binary was assigned 4 cores for the fuzzing engine, with a total of 64 cores for the concolic execution component. The concolic execution pool processed symbolic execution jobs in a first-in-first-out queue as traces were requested by the fuzzing nodes when Driller determined that the fuzzers were "stuck". Symbolic execution traces were restricted to a one-hour period and a 4 gigabyte memory limit to avoid resource exhaustion from analyzing large traces.

We will discuss several different facets of our evaluation of Driller. We will start by discussing the results of the three experiments in terms of Driller's contribution to the number of vulnerabilities that we were able to find in the dataset. Next, we will discuss Driller's contribution in terms of code coverage over existing techniques. Finally, we will focus on an example application from the CGC dataset for an in-depth case study to discuss how Driller increased code coverage and identified the vulnerability in that application.

D. Vulnerabilities

In this subsection, we will discuss the number of vulnerabilities that were discovered by the three experiments, and frame Driller's contribution in this regard.

The symbolic execution baseline experiment faired poorly on this dataset. Out of the 126 applications, symbolic execution discovered vulnerabilities in only 16.

Out of the 126 Cyber Grand Challenge applications in our experimental dataset, fuzzing proved to be sufficient to discover crashes in 68. Of the remaining 58 binaries, 41 became "stuck" (i.e., AFL was unable to identify any new "interesting" paths, as discussed in Section IV, and had to resort to random input mutation) and 17, despite continuing to find new interesting inputs, never identified a crash.

In Driller's run, the fuzzer invoked the concolic execution component on the 41 binaries that became "stuck". Figure 7 shows the number of times that concolic execution was invoked for these binaries. Of these, Driller's concolic execution was able to generate a total of 101 new inputs for 13 of these applications. Utilizing these extra inputs, AFL was able to recover an additional 9 crashes, bringing the total identified crashes during the Driller experiment to 77, meaning that Driller achieves a 12% improvement over baseline fuzzing in relation to discovered vulnerabilities.

Of course, most of the applications for which crashes were discovered in the Driller experiment were found with the baseline fuzzer. In terms of unique crashes identified by the different approaches, the fuzzer baseline discovered 55 crashes symbolic execution failed to discover. 13 of its vulnerabilities were shared with the symbolic execution baseline. A further 3 symbolic execution baseline vulnerabilities overlap with vulnerabilities recovered by Driller, leaving application

for which the symbolic execution baseline alone found a vulnerability, and leaving 6 applications for which Driller's approach was the only one to find the vulnerability. Essentially, Driller effectively merges and expands on the capabilities offered by baseline fuzzing and baseline concolic execution, achieving more results than both do individually. These results are presented in Figure 5.

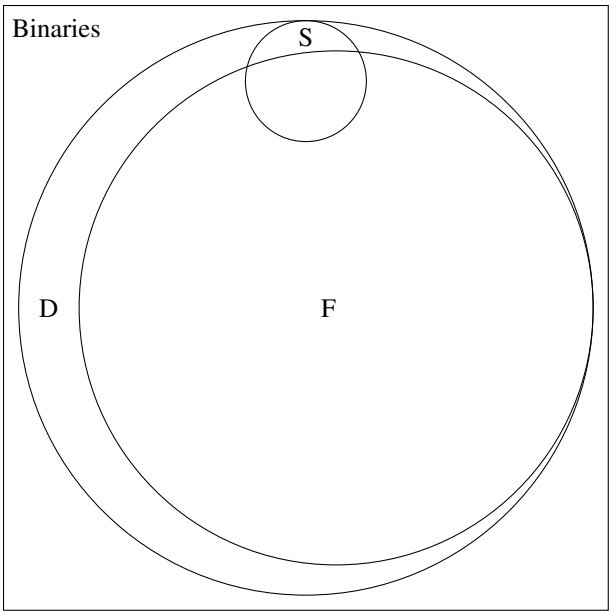
In total, Driller was able to identify crashes in 77 unique applications, an improvement of 6 crashes (8.4%) over the union of the baseline experiments. This is the same number of crashes as identified by the top-scoring team in the competition (and significantly higher than any of the other competitors), in the same amount of time. Without Driller (i.e., with the two baseline approaches), we would not have achieved these results. Note that we are well-aware that the comparison to a participating team is only indicative and it is not meant to be qualitative. The participating team was operating under strict time constraints, with little or no space for errors. Our experiments benefit from additional time to prepare and our techniques could be refined throughout the course of Driller's development.

These results demonstrate that enhancing a fuzzer with selective concolic execution improves its performance in finding crashes. By advancing the state of the art in vulnerability excavation, Driller is able to crash more applications than the union of those found by fuzzing and by symbolic execution separately. While a contribution of 6 unique vulnerabilities might seem low compared to the total number of applications in the CGC qualifying event, these crashes represent vulnerabilities deep in their respective binaries, many of which require multiple concolic execution invocations to penetrate through several compartments.

E. State Transition Coverage

Selective symbolic execution is able to overcome a fundamental weakness of fuzzers when dealing with "magic" constants and other complex input checks. That means that, after the fuzzer is unable to identify new interesting inputs (for example, due to a failure to guess a hash or a magic number), the concolic execution engine can generate an input allowing the fuzzer to continue exploring paths beyond where it had become stuck. This aspect of Driller can be observed in Table I, which shows the breakdown of how state transitions were found during execution. In applications in which the symbolic execution was able to find a new path, fuzzing alone had only found an average of 28.5% of the block transitions.

As expected, the symbolic traces account for only a small amount of new state transitions in these binaries (about 15.1% on average), as the symbolic exploration is limited in scope and reserved mostly for identifying and passing interesting checks. However, the inputs produced by the concolic execution engine help the fuzzing engine in successfully penetrating these state transitions. The fuzzing engine's subsequent modifications of these inputs allow it to find, on average, an additional 56.5% of state transitions. In total, for the applications in which the fuzzer eventually gets stuck and symbolic execution found a new path, 71.6% of the state transitions resulted from the inputs based on those that were generated during symbolic traces. The fact that the small numbers of concolically-contributed inputs result in a much larger set of state transitions



Method	Crashes Found
Fuzzing	68
Fuzzing ∩ Driller	68
Fuzzing ∩ Symbolic	13
Symbolic	16
Symbolic ∩ Driller	16
Driller	77

Fig. 5. The makeup of the experimentation results. The Venn Diagram shows the relative coverage of Basic Fuzzing (AFL), Symbolic Execution, and Driller in terms of finding crashes in the CGC dataset. The circle labeled **F** represents crashes found by fuzzing, **S** represents crashes found by symbolic execution, and **D** represents crashes found by driller. The table presents these results in terms of the relative effectiveness of the different methods and their improvement relative to each other. The attentive reader can see that Driller identifies a super-set of the crashes found by Fuzzing and Symbolic Execution.

that the fuzzer can explore demonstrates that the inputs generated by Driller's concolic execution engine stimulated a much deeper exploration of the application. It is important to keep in mind that this number only applies to 13 of the 41 applications which became "stuck" and were able to have a new path identified by symbolic execution. These percentages are normalized over the total amount of basic blocks that we saw over the course of the experiment, as generating a complete Control Flow Graph statically requires heavyweight static analysis that is outside of the scope of this paper.

As discussed in Section IV, we consider a state transition to be an ordered pair of basic blocks (A,B) where block B is executed immediately following block A. In other words, a state transition is an *edge* in a Control Flow Graph where each node represents a basic block in the program. It is clear that if we find every state transition that we have complete code coverage. Similarly, if we find few state transitions, than we likely have very low coverage. Thus, it is reasonable to use the number of unique state transitions as a measure of code coverage. In Figure 6, we show how Driller improved the basic block coverage over time, by showing how many

additional basic blocks were discovered as a result of Driller, that the fuzzer was unable to find on its own.

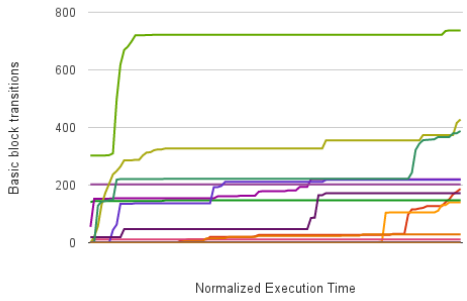


Fig. 6. The number of additional basic blocks found by Driller over time, that the fuzzer was unable to find on its own. Execution time is shown normalized to the execution time of the binary, which varies depending on if/when it crashed. This graph includes the 13 binaries that invoked, and benefited from, concolic execution.

F. Application Component Coverage

A goal of the symbolic traces in Driller is to enable the fuzzer to explore the various compartments in a binary, where the compartments may be separated by complex checks on user input. We expect to see inputs generated by invocations of the concolic tracer correspond to finding new compartments in the application. That is, the inputs generated by the concolic execution engine should enable the fuzzer to reach and explore new areas of code.

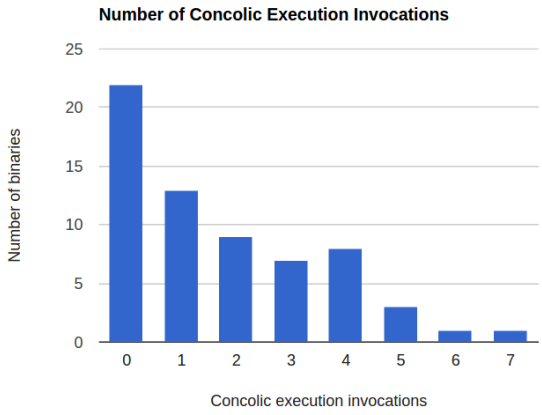


Fig. 7. Graph showing how many times concolic execution was invoked in binaries where fuzzing could not crash the binary on its own.

As shown in Figure 5, 68 of the 126 applications in the data set did not have any difficult checks that needed Driller's symbolic execution. These correspond to applications for which the fuzzing component independently found crashing inputs or for which it never became "stuck". These applications tend to be the ones with simple protocols and fewer complex checks. On the other hand, Driller was able to satisfy at least one difficult check in 13 of the binaries and multiple difficult checks in 4 of the binaries. These compartments are difficult for basic fuzzers to enter because of the specific checks separating them, but solvable by the hybrid approach employed by Driller.

Each invocation of concolic execution has the potential to guide execution to a new compartment in the application. This can be measured by analyzing the basic block coverage of Driller before a fuzzing round gets "stuck" and invokes concolic execution versus the coverage achieved by the subsequent round of fuzzing, after the concolic execution component pushed execution through to the next compartment. We present this in Figure 8, by showing the fraction of basic blocks, normalized to the total number of basic blocks discovered throughout the experiment, for each binary on which concolic execution was invoked, at each stage of the analysis. The graph demonstrates that Driller *does* drive execution into new compartments in applications, allowing the fuzzer to quickly explore a greater amount of code. We present an in-depth example of this for our case study in Section VI-G.

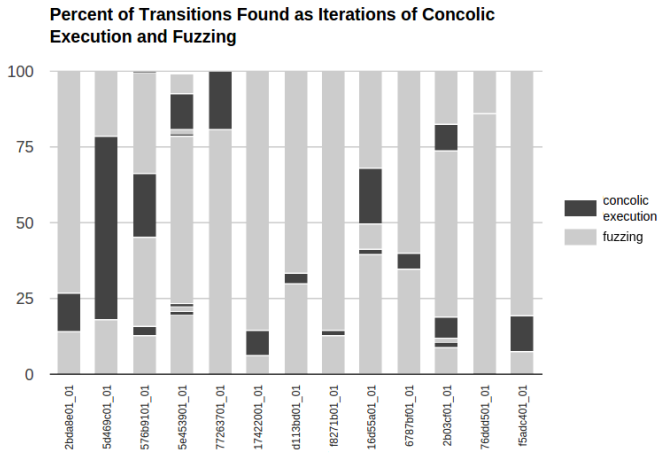


Fig. 8. Graph showing how each invocation of concolic execution lead to more basic block transitions found. Only shown for binaries in which symbolic execution identified additional inputs.

G. Case Study

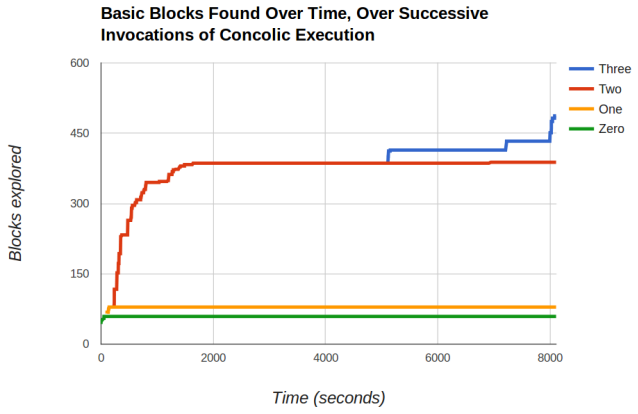


Fig. 9. For the binary 2b03cf01, which Driller crashed in about 2.25 hours, this graph shows the number of basic blocks found over time. Each line represents a different number of invocations of symbolic execution from zero to three invocations. After each invocation of symbolic execution, the fuzzer is able to find more basic blocks.

Type of State Transition	Percentage of discovered blocks across all binaries	Percentage of discovered blocks across binaries where concolic execution found at least one input
Initial Fuzzing Run	84.2	28.4
Identified by Concolic Execution	3.3	15.1
Post-Concolic Fuzzing Runs	12.5	56.5
Total	100	100

TABLE I. BREAKDOWN OF WHAT PERCENTAGE OF DISCOVERED STATE TRANSITIONS WERE FOUND BY WHAT METHOD, AMONG BINARIES WHICH INVOKED CONCOLIC EXECUTION AND BINARIES FOR WHICH CONCOLIC EXECUTION IDENTIFIED AT LEAST ONE INPUT.

```
1  enum {
2      MODE_BUILD = 13980,
3      MODE_EXAMINE = 809110,
4  };
5
6  ...
7
8  RECV(mode, sizeof(uint32_t));
9
10 switch (mode[0]) {
11     case MODE_BUILD:
12         ret = do_build();
13         break;
14     case MODE_EXAMINE:
15         ret = do_examine();
16         break;
17     default:
18         ret = ERR_INVALID_MODE;
19 }
```

Listing 7. The first complex check in the 2b03cf01 application.

This section will focus on a single application to explain Driller's operation in-depth. We will focus on the CGC qualifying event application whose identifier is `2b03cf01`. The interested reader can find the source code for this application on DARPA's github repository [12] under the public name `NRFIN_00017`. Additionally, we present the call graph of this binary, which we will refer to throughout this case study, in Figure 10. This graph demonstrates the performance of successive invocations of Driller's fuzzing and concolic execution components – the nodes discovered by successive fuzzing invocations are drawn in progressively darker colors and the transitions recovered by the concolic execution component are illustrated with differently-drawn edges. The different colors of nodes represent different compartments in the binary.

This application represents a power testing module, in which the client provides the server an electrical design and the server builds a model of the electrical connectivity. This is not a simple binary: it provides a variety of complex functionality to the user and requires properly formatted input, against which there are a number of complex checks.

When Driller fuzzes this binary, the first of these complex checks causes the fuzzer to get stuck almost immediately after finding only 58 basic blocks across a fairly small compartment of the application, consisting of a handful of functions containing initialization code. The fuzzing engine gets stuck on a check on user input. For convenience, the snippet in question, corresponding to node "A" in Figure 10, is reproduced in Listing 7, although, Driller is operating directly on the binary code.

Looking at the source code, we see that the two primary commands called from the main loop require the user to give a specific 32-bit number to select a "mode of operation". To call the function *do_build()*, the user must provide the number 13980, and to call the function *do_examine()*, the user must provide the number 809110. Although, these checks appear simple to a human, a fuzzer must essentially brute force them. Thus, the chance that the fuzzer will guess these magic numbers is minuscule, and, as a result, the fuzzing component gets stuck.

After the fuzzer is unable to identify new interesting paths, Driller invokes the concolic execution component to trace the inputs that the fuzzer has collected thus far, and find new state transitions. Driller finds inputs which will drive execution to both of the aforementioned functions, and returns them to the fuzzer for exploration. Again, the fuzzer gets stuck fairly quickly, this time at node "B" in Figure 10 at another complex check. Driller's concolic execution engine is invoked a second time, generating enough new inputs to pass these checks. From this point, the fuzzer is able to find 271 additional basic blocks within a large compartment of the application that processes generic input which, for this application, consists of parsing code relating to analysis of the user-provided electrical design. Eventually, the fuzzer finds all of the interesting paths that it can in that compartment and decides that it is not making further progress, leading to another invocation of Driller's concolic execution engine.

This time, Driller finds 74 new basic blocks and generates inputs that reach them by successfully passing checks on the input that the fuzzer had not previously satisfied. These additional basic blocks (represented by the black nodes in Figure 10) comprise the functionality of adding specific circuit components. For the interested reader, Listing 9 presents one of the functions that contains specific checks against user input with which fuzzers have trouble. Input representing these components must adhere to an exact specification of a circuit component, and the checks of these specifications is what the third invocation of Driller's concolic execution engine finds. These constants that are used in this function's checks are defined in the code reproduced in Listing 8. A fuzzer cannot guess these constants without exhausting a huge search space, as they are specific values of 32-bit integers. Driller's symbolic execution, however, can find these constants easily, since the comparisons in the code produce easy-to-solve-for conditions on paths taking these branches.

```
1 typedef enum {
2   FIFTEEN_AMP = 0x0000000f,
3   TWENTY_AMP = 0x00000014,
4 } CIRCUIT_MODELS_T;
```

Listing 8. An enum definition with explicit constants. In order to guess these constants, these specific values must be guessed from a search space of 2^{32} numbers.

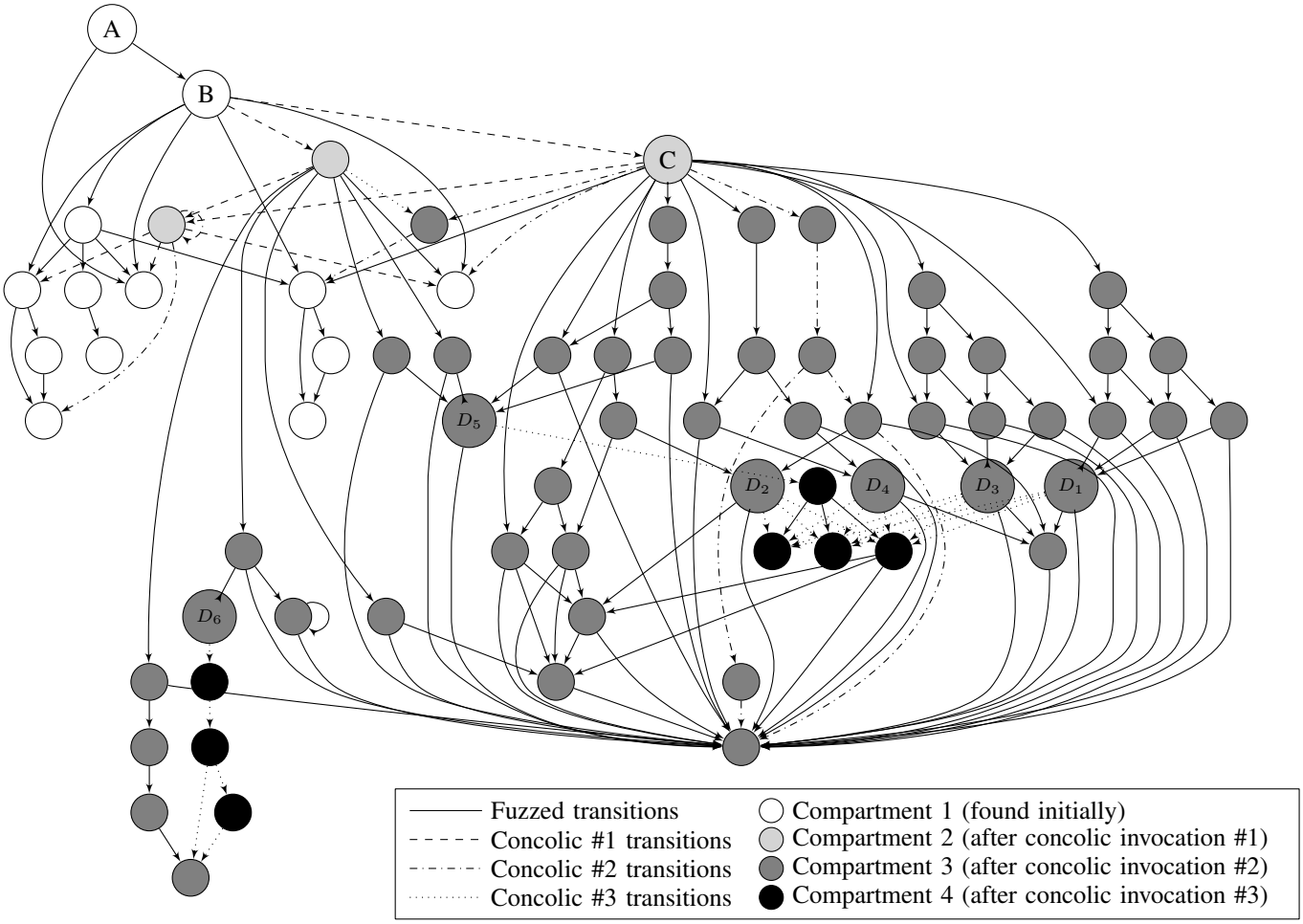


Fig. 10. Graph visualizing the progress made by Driller in discovering new compartments. Each node is a function; each edge is a function call, but return edges are excluded to maintain legibility. Node "A" is the entry point. Node "B" contains a magic number check that requires the symbolic execution component to resolve. Node "C" contains another magic number check.

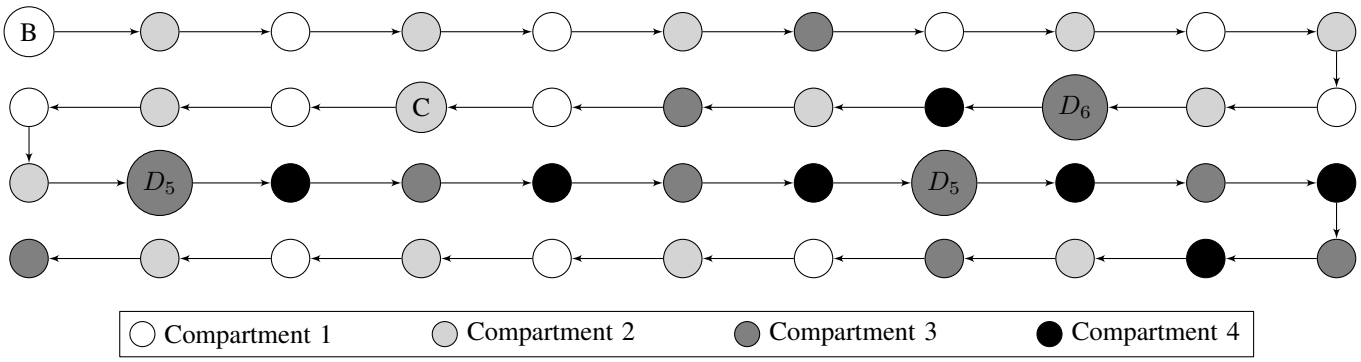


Fig. 11. The sequence of compartments through which execution flows for a trace of the crashing input for CGC application 2b03cf01. Driller's ability to "break into" the fourth compartment (represented by the black nodes) was critical for generating the crashing input. The generated, derandomized crashing input was "A\x00\x00\x00\x00\x00\x00\x00\x9c6\x00\x00\x18\x04\x00\x00\x18'\x00\x00A\x00\x00\x00\x00\x00\x00\x00\x9c6\x00\x00\x19\x04\x00 \x00\x14\x00\x00\x00A\x00\xf8\xff\xff\xec\x00d\x96X\x0c\x00\x06\x08\x00\x00\x10\x00\x00\x00A\x00\x00\x00\x00\x00\x00\xfb\x96X \x0c\x00\x02\x08\x00\x00\x18'\x00\x00A\x00\xebA\x00\x00d\x96X\x0c\x00\x06". The full exploit specification, conforming to the DARPA CGC exploit specification format and accounting for randomness, is available in Appendix A.

```
1  int8_t get_new_breaker_by_model_id
       (CIRCUIT_MODELS_T model_id, breaker_t *
       breaker_space, uint8_t breaker_space_idx) {
2    int8_t res = SUCCESS;
3    switch(model_id) {
4      case FIFTEEN_AMP:
5        create_breaker(15,
               breaker_space, breaker_space_idx);
6        break;
7      case TWENTY_AMP:
8        create_breaker(20,
               breaker_space, breaker_space_idx);
9        break;
10     default:
11       // invalid model_id
12       res = -1;
13   }
14   return res;
15 }
```

Listing 9. A function with a switch statement testing user input against a number of specific values

Driller's new input is then passed back to the fuzzer in order to quickly assess the new coverage generated by the change. Listing 10 shows some of the code that executes for the first time as a result of the new input. The user input that this new component processes is no longer specific, but general, making it suitable for the fuzzer. From this point on, the fuzzer continues to mutate these inputs until it triggers a vulnerability caused by a missing sanitization check in the application.

```
1  static void create_breaker
       (uint8_t amp_rating, breaker_t *
       breaker_space, uint8_t breaker_space_idx) {
2    breaker_space->id = breaker_space_idx;
3    breaker_space->amp_rating = amp_rating;
4    breaker_space->outlets = list_create_dup();
5    if (breaker_space->outlets
         == NULL) {_terminate(ERRNO_ALLOC);}
6  }
```

Listing 10. Code executed as a result of passing the specific check

In a semantic sense, the vulnerability involves initializing a new breaker object in the circuit the user creates. Later on, the circuit will be tested for connectivity, among other things, and component-specific logic will be invoked depending on the materials of which the circuit is composed. Satisfying the check to add a breaker will expand the bug-searching coverage to include breaker-specific code. Triggering this vulnerability requires the inclusion, in the provided circuit diagram, of specifically crafted breaker components. The inputs required to trigger the creation of these components are what Driller recovers in the third concolic execution invocation, and the final fuzzing invocation mutates them enough to trigger the vulnerable edge case.

The final path taken by the crashing input is shown in Figure 11. Starting at the entry point, this path goes through progressively harder-to-reach compartments (represented by the different colors of the nodes) until the condition to trigger the edge is created. This binary was not crashed in either baseline experiment – the unaided fuzzer was never able to reach compartments of the code "protected" by the complex checks, and the symbolic exploration engine experienced an almost immediate path explosion in the input-processing code. By combining the merits of fuzzing and concolic execution, Driller found a crash for this binary in approximately two hours.

We present the amplification in basic block coverage that each concolic execution invocation produces in this binary, plotted over time, in Figure 9.

VII. Discussion

Driller carries out a *unified* analysis by leveraging both symbolic execution and fuzzing. This allows Driller to address some of the drawbacks of each analysis with its complement. In this section, we will discuss the limitations of Driller and future directions of research to further augment automated vulnerability extraction.

A. Limitations

Both a benefit and pitfall of Driller is its borrowing of state-space interpretation from AFL. AFL represents state simply by tracking state-transition tuples to rough "hit counts" (how many times the state-transition was encountered). This moderately light representation of state is what allows AFL to be so efficient as each path's state is only defined by the collection of state-transition tuples it encountered combined with how many times they were encountered. Driller uses this same data structure to determine which state transitions are worth solving for. We provide an example of how this can limit Driller in Listing 11

```
1  int static_strcmp(char *a, char *b){
2      for (;*a;a++,b++) {
3          if (*a != *b)
4              break;
5      }
6
7      return *a - *b;
8  }
9
10 int main(void) {
11   read(0, user_command, 10);
12
13   if (static_strcmp
         ("first_cmd", user_command) == 0) {
14       cmd1();
15   }
16     else if (static_strcmp
           ("second_cmd", user_command) == 0) {
17       cmd2();
18     }
19     else if (static_strcmp
           ("crash_cmd", user_command) == 0) {
20       abort();
21     }
22
23   return 0;
24 }
```

Listing 11. An example of minimal state representation limiting discovery of new state transitions.

This listing demonstrates a state-transition which occurs in multiple command handlers. Since each branch relies on `static_strcmp`, AFL itself will not be able to distinguish between state-transitions inside different invocations of `static_strcmp`. Driller uses the same metric to determine which state-transitions need to be solved. As such, Driller will not try to solve for the `if` statement on line 3 more than once, even though it is used for different comparisons. Additionally, inputs which have one or two additional matching characters would not be considered interesting by AFL. Of course if the

entire string was discovered by Driller, AFL would find it interesting and adopt it. Driller attempts to mitigate the effects of this problem with the symbolic explorer stub (described in V-D3) invoked at each new state transition. However, we believe this is an imperfect solution and ultimately a better representation of state might be required.

Another limitation of Driller is the case when user input is treated as *generic* input in one component and *specific* input in another. Consider the program presented in Listing 12.

This application reads a command and a hash from the user and verifies the hash. This compartment, spanning lines 1 through 11, treats the command as generic input and the hash as specific input. After this, however, the application checks, in multiple stages, that the provided command was "`CRASH!!`". Fundamentally, this reclassifies the `user_command` as specific input, as it must be matched exactly. This triggers a case that reduces Driller to a symbolic execution engine, as explained below.

```
1  int main(void) {
2    char user_command[10];
3    int user_hash;
4
5    read(0, user_command, 10);
6    read(0, user_hash, sizeof(int));
7
8    if (user_hash != hash(user_command)) {
9      puts("Hash mismatch!");
10     return 1;
11   }
12
13   if (strncmp("CRASH", user_command, 5) == 0) {
14     puts("Welcome to compartment 3!");
15     if (user_command[5] == '!') {
16       path_explosion_function();
17       if (user_command[6] == '!') {
18         puts("CRASHING");
19         abort();
20       }
21     }
22   }
23
24   return 0;
25 }
```

Listing 12. An example of input being used as generic input in one place and specific input in another. A crashing input for this binary is "CRASH!!" followed by its hash.

Passing through the first stage, into compartment 3, is straightforward – Driller's concolic execution engine will identify an input that starts with "`CRASH`" and its corresponding hash (as this is a forward-calculation of a hash, there is no concern with having to "crack" the hash; Driller merely needs to calculate it). However, after this, the fuzzer will no longer function for exploring this compartment. This is because any random mutations to either the hash or the input will likely cause execution to fail to proceed from compartment 1. Thus, the fuzzer will quickly get stuck, and Driller will invoke the concolic execution engine again. This invocation will guide Driller to compartment 4, on line 16, and hand execution back to the fuzzer. However, the fuzzer will again fail to proceed, decide that it is stuck, and trigger concolic execution.

This cycle will continue, making the fuzzing component useless and essentially reducing Driller to symbolic exploration. Worse, in this application, compartment 4 calls a function (`path_explosion_function`) that causes a path explosion. Without the mitigating effects of its fuzzing engine, Driller is unable to reach compartment 5 (lines 18 and 19) and trigger the bug.

This represents a limitation in Driller: in certain cases, the fuzzing component can become effectively disabled, robbing Driller of its advantage. A potential future step in mitigating this issue is the ability to generate "semi-symbolic" fuzzing input. For example, the concolic engine might pass a set of constraints to the fuzzer to ensure that the inputs it generated conform to some specification. This would take advantage of the concept of *generational fuzzing* [18] to create "input generators" to aid the fuzzer in reaching and exploring application compartments.

The limitation exemplified by Listing 12 shows how a specific input can prevent the fuzzer from effectively mutating the generic input. However, for other types of specific input, even with multiple components, AFL can still fuzz the deeper components. Even in the most difficult cases, such as hash checks, Driller will still be able to mutate any input that is unrelated to the hash, such as input after the hash is checked. We do expect some decrease in performance after Driller has found multiple components. This is because AFL has no knowledge of the constraints from the symbolic execution engine, so there will be a fraction of the fuzzing cycles wasted trying to mutate specific inputs.

VIII. Conclusion

In this paper, we presented Driller, a tool that combines the best of dynamic fuzzing and concolic execution to efficiently find bugs buried in a binary. We introduce the concept of a *compartment* of a binary, which largely separate functionality and code. Within Driller, fuzzing provides a fast and cheap overview of a compartment, effectively exploring loops and simple checks, but often fails to transition *between* compartments. Selective concolic execution gets into state explosions when considering loops and inner checks, but is highly effective at finding paths between compartments of a binary. By combining these two techniques, where each individually fails, Driller is able to explore a greater space of functionality within the binary.

We evaluated Driller on 126 binaries from the DARPA Cyber Grand Challenge Qualifying Event. Driller found 77 crashes, a substantial improvement over basic fuzzing's 68 crashes. We believe the technique shows promise for general-purpose bug-finding in all categories of binaries.

Acknowledgments

We would like to thank all contributors to the DARPA Cyber Grand Challenge organization (for providing an excellent testing ground for our tool), Michal Zalewski (his public tool, AFL, and documentation proved immensely useful), the contributors of angr, and of course, all our fellow Shellphish CGC team members, *donfos* in particular. This material is based on research sponsored by DARPA under agreement number N66001-13-2-4039. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work is also based upon work supported by SBA Research. The views

and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the U.S. Government, or SBA Research.

References

[1] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1083–1094. ACM, 2014.

[2] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] S. Bucur. *Improving Scalability of Symbolic Execution for Software with Complex Environment Interfaces*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2015.

[4] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[6] G. Campana. Fuzzgrind: un outil de fuzzing automatique. In *Actes du 7ème symposium sur la sécurité des technologies de linformation et des communications (SSTIC)*, 2009.

[7] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song. Transformation-aware exploit generation using a HI-CFG. Technical report, UCB/EECS-2013-85, 2013.

[8] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[9] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 48. ACM, 2013.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278. ACM, 2011.

[11] DARPA. Cyber Grand Challenge. http://cybergrandchallenge.com.

[12] DARPA. Cyber Grand Challenge Challenge Repository. https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challenges.

[13] J. DeMott. Understanding how fuzzing relates to a vulnerability like Heartbleed. http://labs.bromium.com/2014/05/14/understanding-how-fuzzing-relates-to-a-vulnerability-like-heartbleed/.

[14] C. Details. Vulnerability distribution of CVE security vulnerabilities by type. http://www.cvedetails.com/vulnerabilities-by-types.php.

[15] W. Drewry and T. Ormandy. Flayer: Exposing application internals. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2007.

[16] D. Engler and D. Dunbar. Under-constrained execution: Making automatic code destruction easy and scalable. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2007.

[17] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[18] P. Garg. Fuzzing - mutation vs. generation. http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 40, pages 213–223. ACM, 2005.

[20] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[21] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the USENIX Security Symposium*, 2013.

[22] LegitBS. DEFCON Capture the Flag. https://legitbs.net/.

[23] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[24] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 42, pages 89–100. ACM, 2007.

[25] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos. The BORG: Nanoprobing binaries for buffer overreads. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 2015.

[26] B. S. Pak. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. Master's thesis, School of Computer Science, Carnegie Mellon University, May 2012.

[27] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the USENIX Security Symposium*, 2015.

[28] Secunia. Resources vulnerability review 2015. http://secunia.com/resources/vulnerability-review/introduction/.

[29] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[30] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[31] M. Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

Appendix A
Exploit Specification for Case Study

For the curious reader, we reproduce the exact crashing input, specified in the Cyber Grand Challenge Proof of Vulnerability format in Listing 13.

```
1  <?xml version="1.0" standalone="no" ?>
2  <!DOCTYPE pov SYSTEM "/usr/share/cgc-docs/replay.dtd">
3  <pov><cbid>2b03cf01_01</cbid><replay>
4    <read><length>1</length><assign><var>stdout0</var><slice
         begin="0" /></assign></read> <read><length>7</length></read>
5    <write><var>
         stdout0</var></write><write><data>\x00\x00\x00\x00\x00\x00\x00
         \x9c6\x00\x00\x18\x04\x00\x00\x18\x27\x00\x00</data></write>
6    <read><length>1</length><assign><var>stdout8</var><slice
         begin="0" /></assign></read> <read><length>7</length></read>
7    <write><var>stdout8</var></write><write><data
         >\x00\x00\x00\x00\x00\x00\x00\x9c6\x00\x00\x19\x04\x00\x00\
         x14\x00\x00\x00</data></write><read><length>37</length></read>
8    <read><length>1</length><assign><var>stdout16</var><slice
         begin="0" /></assign></read> <read><length>7</length></read>
9    <write><var>stdout16</var></write><write
         ><data>\x00\xf8\xff\xff\xec\x00d\x96X\x0c\x00\x06\x08\x00\x00\
         x10\x00\x00\x00</data></write><read><length>37</length></read>
10   <read><length>1</length><assign><var>stdout24</var><slice
         begin="0" /></assign></read> <read><length>7</length></read>
11   <write><var>stdout24</var></
         write><write><data>\x00\x00\x00\x00\x00\x00\xfb\x96X\x0c\x00\
         x02\x08\x00\x00\x18\x27\x00\x00</data></write></replay> </pov>
```

Listing 13. A Proof of Vulnerability for the case study application.

This input can be run on a Cyber Grand Challenge machine by executing the command shown in Listing 14.

```
1  cb-test --debug --should_core --cb ./NRFIN_00017
       --xml ./POV_FROM_LISTING.xml --directory .
```

Listing 14. The command to test the provided Proof of Vulnerability.