(State of) The Art of War: Offensive Techniques in Binary Analysis

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher,
John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, Giovanni Vigna
UC Santa Barbara
{yans,fish,salls,stephens,mario,dutcher,jmg,slipper,christophe,chris,vigna}@cs.ucsb.edu

***Abstract*—Finding and exploiting vulnerabilities in binary code is a challenging task. The lack of high-level, semantically rich information about data structures and control constructs makes the analysis of program properties harder to scale. However, the importance of binary analysis is on the rise. In many situations binary analysis is the only possible way to prove (or disprove) properties about the code that is *actually* executed.**

In this paper, we present a binary analysis framework that implements a number of analysis techniques that have been proposed in the past. We present a systematized implementation of these techniques, which allows other researchers to compose them and develop new approaches. In addition, the implementation of these techniques in a unifying framework allows for the direct comparison of these approaches and the identification of their advantages and disadvantages. The evaluation included in this paper is performed using a recent dataset created by DARPA for evaluating the effectiveness of binary vulnerability analysis techniques.

Our framework has been open-sourced and is available to the security community.

I. Introduction

Despite the rise of interpreted languages and the World Wide Web, binary analysis has remained an extremely important topic in computer security. There are several reasons for this. First, interpreted languages are either interpreted by binary programs or Just-In-Time (JIT) compiled down to binary code. Second, "core" OS constructs and performance-critical applications are still written in languages (usually, C or C++) that compile down to binary code. Third, the rise of the Internet of Things is powered by devices that are, in general, very resource-constrained. Without cycles to waste on interpretation or Just-In-Time compilation, the firmware of these devices tends to be written in languages (again, usually C) that compile to binary.

Unfortunately, many low-level languages provide few security guarantees, often leading to vulnerabilities. For example, buffer overflows stubbornly remain as one of the most-common software flaws despite a concerted effort to develop technologies to mitigate such vulnerabilities. Worse, the wider class of "memory corruption vulnerabilities", the vast majority of which also stem from the use of unsafe languages, make up a substantial portion of the most common vulnerabilities [2]. This problem is not limited to software on general-purpose computing devices: remotely-exploitable vulnerabilities have been discovered in devices ranging from smartlocks, to pacemakers, to automobiles [10].

Another important aspect to consider is that compilers and tool chains are not bug-free. Properties that were proven by analyzing the source code of a program may not hold after the very same program has been compiled [56]. This happens in practice: recently, a malicious version of Xcode, known as Xcode Ghost [3], silently infected over 40 popular iOS applications by inserting malicious code at compile time, compromising the devices of millions of users. These vulnerabilities have serious, real-world consequences, and discovering them before they can be abused is paramount. To this end, the security research community has invested a substantial amount of effort in developing analysis techniques to identify flaws in binary programs [55]. Such "offensive" (because they find "attacks" against the analyzed application) analysis techniques vary widely in terms of the approaches used and the vulnerabilities targeted, but they suffer from two main problems.

First, many implementations of binary analysis techniques begin and end their existence as a research prototype. When this happens, much of the effort behind the contribution is wasted, and future researchers must often start from scratch in terms of implementation of work based upon these approaches. This startup cost discourages progress: every week spent re-implementing previous techniques is one less week devoted to developing novel solutions.

Second, as a consequence of the amount of work required to reproduce these systems and their frequent unavailability to the public, replicating their results becomes impractical. As a result, the *applicability* of individual binary analysis techniques relative to other techniques becomes unclear. This, along with the inherent complexity of modern operating systems and the difficulty to accurately and consistently model the applications' interaction with their environment, makes it extremely difficult to establish a common ground for comparison. Where comparisons do exist, they tend to compare systems with different underlying implementation details and different evaluation datasets.

In an attempt to mitigate the first issue, we have created `angr`, a binary analysis framework that integrates many of the state-of-the-art binary analysis techniques in the literature. We did this with the goal of systematizing the field and encouraging the development of next-generation binary analysis techniques by implementing, in an accessible, open, and usable way, effective techniques from current research efforts so that they can be easily compared with each other. `angr` provides

building blocks for many types of analyses, using both static and dynamic techniques, so that proposed research approaches can be easily implemented and their effectiveness compared to each other. Additionally, these building blocks enable the *composition* of analyses to leverage their different strengths.

Over the last year, a solution has also been introduced to the second problem, aimed towards comparing analysis techniques and tools, with research reproducibility in mind. Specifically, DARPA organized the Cyber Grand Challenge, a competition designed to explore the current state of automated binary analysis, vulnerability excavation, exploit generation, and software patching. As part of this competition, DARPA wrote and released a corpus of applications that are specifically designed to present realistic challenges to automated analysis systems and produced the *ground truth* (labeled vulnerabilities and exploits) for these challenges. This dataset of binaries provides a perfect test suite with which to gauge the relative effectiveness of various analyses that have been recently proposed in the literature. Additionally, during the DARPA CGC qualifying event, teams around the world fielded automated binary analysis systems to attack and defend these binaries. Their results are public, and provide an opportunity to compare existing offensive techniques in the literature against the best that the competitors had to offer[1].

Our goal is to gain an understanding of the relative efficacy of modern offensive techniques by implementing them in our binary analysis system. In this paper, we detail the implementation of a next-generation binary analysis engine, `angr`. We present several offensive analyses that we developed using these techniques (specifically, replications of approaches currently described in the literature) to reproduce results in the fields of vulnerability discovery, exploit replaying, automatic exploit generation, compilation of ROP shellcode, and exploit hardening. We also describe the challenges that we overcome, and the improvements that we achieved, by combining these techniques to augment their capabilities. By implementing them atop a common analysis engine, we can explore the differences in effectiveness that stem from the theoretical differences behind the approaches, rather than implementation differences of the underlying analysis engines. This has enabled us to perform a comparative evaluation of these approaches on the dataset provided by DARPA.

In short, we make the following contributions:

1) We reproduce many existing approaches in offensive binary analysis, in a single, coherent framework, to provide an understanding of the relative effectiveness of current offensive binary analysis techniques.
2) We show the difficulties (and solutions to those difficulties) of combining diverse binary analysis techniques and applying them on a large scale.
3) We open source our framework, `angr`, for the use of future generations of research into the analysis of binary code.

[1]The top-performing 7 teams each won a prize of $750,000$ USD. We expect that, with this motivation, the teams fielded the best analyses that were available to them.

II. Automated Binary Analysis

Researchers have been striving toward automated binary analysis techniques for many years. However, despite recent advances in this field, such systems are challenging to develop and deploy in the real world. This is because, depending on the technique in question, there are serious limitations that must be overcome to perform automated analysis on real-world software. In this section, we will touch on the challenges of automated analysis and discuss why the DARPA Cyber Grand Challenge contest can provide a meaningful way to compare different analysis approaches.

A. Trade-offs

It is not hard to see why binary analysis is challenging: in a sense, asking "will it crash?" is analogous to asking "will it stop?", and any such analysis quickly runs afoul of the halting problem [32]. Program analyses, and especially offensive binary analyses, tend to be guided by carefully balanced theoretical trade-offs to maintain feasibility. There are two main areas where such trade-offs must be made:

Replayability. Bugs are not all created equal. Depending on the trade-offs made by the system, bugs discovered by a given analysis might not be *replayable*. This boils down to the *scope* that an analysis operates on. Some analyses execute the whole application, from the beginning, so they can reason about what *exactly* needs to be done to trigger a vulnerability. Other systems analyze individual pieces of an application: they might find a bug in a specific module, but cannot reason about how to *trigger* the execution of that module, and therefore, cannot automatically *replay* the crash.

Semantic insight. Some analyses lack the ability to reason about the program in semantically meaningful ways. For example, a dynamic analysis might be able to trace the code executed by an application but not understand *why* it was executed or *what parts* of the input caused the application to act in that specific way. On the other hand, a symbolic analysis that can determine the specific bytes of input responsible for certain program behaviors would have a higher semantic understanding.

In order to offer replayability of input or semantic insight, analysis techniques must make certain trade-offs. For example, high replayability is associated with low coverage. This is intuitive: since an analysis technique that produces replayable input must understand how to reach any code that it wants to analyze, it will be unable to analyze as much code as an analysis that does not. On the other hand, without being able to replay triggering inputs to validate bugs, analyses that do not prioritize bug replayability suffer from a high level of *false positives* (that is, flaw detections that do not represent actual vulnerabilities). In the absence of a replayable input, these false positives must be filtered by heuristics which can, in turn, introduce false negatives.

Likewise, in order to achieve semantic insight into the program being analyzed, an analysis must store and process a

large amount of data. A semantically-insightful dynamic analysis, for example, might store the conditions that must hold in order for specific branches of a program to be taken. On the other hand, a static analysis tunes semantic insight through the chosen data domain – simpler data domains (i.e., by tracking *ranges* instead of actual values) represent less semantic insight.

Analyses that attempt both reproducibility and a high semantic understanding encounter issues with scalability. Retaining semantic information for the entire application, from the entry point through all of the actions it might take, requires a processing capacity conceptually identical to the resources required to execute the program under all possible conditions. Such analyses do not scale, and, in order to be applicable, must discard information and sacrifice *soundness* (that is, the guarantee that all potential vulnerabilities will be discovered).

Aside from these fundamental challenges, there are also implementation challenges. The biggest one of these is the *environment model*. Any analysis with a high semantic understanding must model the application's interaction with its environment. In modern operating systems, such interactions are incredibly complex. For example, modern versions of Linux include over three hundred system calls, and for an analysis system to be complete, it must model the effects of all of them.

Example. To demonstrate the various challenges of binary analysis, we provide a concrete example of a program with multiple vulnerabilities in Listing 1. For clarity and space reasons, this example is simplified, and it is meant only to expose the reader to ideas that will be discussed later in the paper.

Observe the three calls to `memcpy`: the ones on lines 10 and 30 will result in buffer overflows, while the one on line 16 will not. However, depending on the amount of information tracked, a static analysis technique might report all three calls to `memcpy` as potential bugs, including the one on line 16, because it would not have the information to determine that no buffer overflow is possible. Additionally, while the report from a static analysis would include the locations of these bugs, it will not provide inputs to trigger them.

A dynamic technique, such as fuzzing, has the benefit of creating actionable inputs that will trigger any bugs found. On the other hand, simple fuzzing techniques typically only find shallow bugs and fail to pass through code requiring precisely crafted input. In Listing 1, dynamic techniques will have difficulty finding the bug at line 10, because it requires a specific input for the condition to be satisfied. However, because the overflow on line 30 can be triggered through random testing, fuzzing techniques should be able to find an input which triggers the bug.

To find the bug on line 10, one could introduce an abstract data model to reason about many possible inputs at once. One such approach is Dynamic Symbolic Execution (DSE). However, dynamic symbolic techniques, while powerful, suffer from the "path explosion problem", where the number of paths grows exponentially with each branch and quickly becomes intractable. A symbolic execution will detect the bug on line 10 and generate an input for it using a constraint solver. Additionally, it should be able to prove that the `memcpy` on line 16 cannot overflow. However, the execution will likely not be able to find the bug at line 30, as there are too many potential paths which do not trigger the bug.

```
1  int main(void) {
2    char buf[32];
3
4    char *data = read_string();
5    unsigned int magic = read_number();
6
7    // difficult check for fuzzing
8    if (magic == 0x31337987) {
9      // buffer overflow
10     memcpy(buf, data, 100);
11   }
12
13   if (magic < 100 && magic % 15 == 2 &&
14       magic % 11 == 6) {
15     // Only solution is 17; safe
16     memcpy(buf, data, magic);
17   }
18
19   // Symbolic execution will suffer from
20   // path explosion
21   int count = 0;
22   for (int i = 0; i < 100; i++) {
23     if (data[i] == 'Z') {
24       count++;
25     }
26   }
27
28   if (count >= 8 && count <= 16) {
29     // buffer overflow
30     memcpy(buf, data, count*20);
31   }
32
33   return 0;
34 }
```

Listing 1: An example where different techniques will report different bugs.

B. The DARPA Cyber Grand Challenge

In October of 2013, DARPA announced the DARPA Cyber Grand Challenge [23]. Like DARPA Grand Challenges in other fields (such as robotics and autonomous vehicles), the CGC pits teams from around the world against each other in a competition in which all participants must be autonomous programs. A participant's goal in the Cyber Grand Challenge is straight-forward: their system must autonomously identify, exploit, and patch vulnerabilities in the provided software. Millions of dollars in prize money were announced: the top 7 teams to complete the CGC Qualifying Event (held in June, 2015) received $750,000$ USD, and the top 3 teams in the CGC Final Event (held in August, 2016) will receive $2,000,000$ USD, $1,000,000$ USD, and $750,000$ USD, respectively.

The organizers of the Cyber Grand Challenge have put much thought into designing a competition for automated binary analysis systems. For example, they addressed the *environment model* problem by creating a new OS specifically for the CGC: the DECREE OS. DECREE is an extremely simple operating system with just 7 system calls: `transmit`, `receive`, and `waitfd` to send, receive, and wait for data over file descriptors, `random` to generate random data,

`allocate` and `deallocate` for memory management, and `terminate` to exit.

Despite the simple environment model, the binaries provided by DARPA for the CGC have a wide range of complexity. They range from 4 kilobytes to 10 megabytes in size, and implement functionality ranging from simple echo servers, to web servers, to image processing libraries. DARPA has open-sourced all of the binaries used in the competition thus far, complete with proof-of-concept exploits and write-ups about the vulnerabilities [24].

Because the simple environment model makes it feasible to accurately implement and evaluate (on a large scale) binary analysis techniques, we use the DARPA CGC samples as our dataset for the comparative evaluations in this paper.

C. Comparative Analysis of CGC Binaries

Offensive binary analyses use different underlying techniques to reason about the application that is being processed. For example, they may analyze data over different *domains* or utilize different levels of interaction with the application being tested. In the next two sections, we survey the current state of the art, and choose several analyses for in-depth evaluation in the rest of the paper. We focus specifically on analyses whose goals are to identify and exploit flaws in binary software (for example, memory safety violation identification using symbolic execution), as opposed to the more general binary analysis techniques on which those are based (in this case, symbolic execution itself).

III. Background: Static Vulnerability Discovery

Static techniques reason about a program without executing it. Usually, a program is interpreted over an *abstract domain*. Memory locations containing bits of ones and zeroes contain other abstract entities (at the familiar end, this might simply be integers, but, as we explain below, these can include more abstract constructs). Additionally, program constructs such as the layout of memory, or even the execution path taken, may be abstracted as well.

Here, we split static analyses into two paradigms: those that model program properties as graphs (i.e., a *control-flow graph*) and those that model the data itself.

Static vulnerability identification techniques have two main drawbacks, relating to the trade-offs discussed in Section II-A. First, the results are not *replayable*: detection by static analysis must be verified by hand, as information on *how* to trigger the detected vulnerability is not recovered. Second, these analyses tend to operate on simpler data domains, reducing their *semantic insight*. In short, they over-approximate: while they can often authoritatively reason about the *absence* of certain program properties (such as vulnerabilities), they suffer from a high rate of false positives when making statements regarding the *presence* of vulnerabilities.

A. Recovering Control Flow

The recovery of a *control-flow graph* (CFG), in which the nodes are basic blocks of instructions and the edges are possible control flow transfers between them, is a pre-requisite for almost all static techniques for vulnerability discovery.

Control-flow recovery has been widely discussed in the literature [21], [33], [34], [50], [58], [59]. CFG recovery is implemented as a recursive algorithm that disassembles and analyzes a basic block (say, B_a), identifies its possible *exits* (i.e., some successor basic block such as B_b and B_c) and adds them to the CFG (if they have not already been added), connects B_a to B_b and B_c, and repeats the analysis recursively for B_b and B_c until no new exits are identified. CFG recovery has one fundamental challenge: indirect jumps. Indirect jumps occur when the binary transfers control flow to a target represented by a value in a register or a memory location. Unlike a *direct* jump, where the target is encoded into the instruction itself and, thus, is trivially resolvable, the target of an indirect jump can vary based on a number of factors. Specifically, indirect jumps fall into several categories:

Computed. The target of a computed jump is determined by the application by carrying out a calculation specified by the code. This calculation could further rely on values in other registers or in memory. A common example of this is a jump table: the application uses values in a register or memory to determine an index into a jump table stored in memory, reads the target address from that index, and jumps there.

Context-sensitive. An indirect jump might depend on the context of an application. The common example is `qsort()` in the standard C library – this function takes a *callback* that it uses to compare passed-in values. As a result, some of the jump targets of basic blocks inside `qsort()` depend on its caller, as the caller provides the callback function.

Object-sensitive. A special case of context sensitivity is object sensitivity. In object-oriented languages, object polymorphism requires the use of virtual functions, often implemented as *virtual tables* of function pointers that are consulted, at runtime, to determine jump targets. Jump targets thus depend on the type of object passed into the function by its callers.

Different techniques have been designed to deal with different types of indirect jumps, and we will discuss the implementation of several of them in Section VII. In the end, the goal of CFG recovery is to *resolve* the targets of as many of these indirect jumps as possible, in order to create a CFG. A given indirect jump might resolve to a *set* of values (*i.e.*, all of the addresses in a jump table, if there are conditions under which their use can be triggered), and this set might change based on both object and context sensitivity. Depending on how well jump targets are resolved, the CFG recovery analysis has two properties:

Soundness. A CFG recovery technique is *sound* if the set of all potential control flow transfers is represented in the graph generated. That is, when an indirect jump is resolved to a *subset* of the addresses that it can actually target, the soundness of the graph decreases. If a potential

target of a basic block is missed, the block it targets might never be seen by the CFG recovery algorithm, and any direct and indirect jumps made by that block will be missed as well. This has a cumulative effect: the failure to resolve an indirect jump might severely reduce the completeness of the graph. Soundness can be thought of as the *true positive* rate of indirect jump target identification in the binary.

Completeness. A complete CFG recovery builds a CFG in which all edges represent actually possible control flow transfers. If the CFG analysis errs on the side of *completeness*, it will likely contain edges that cannot really exist in practice. Completeness can be thought of as the inverse of the *false positive* rate of indirect jump target identification.

A CFG recovery analysis that produces an empty graph would be considered *complete*, and an analysis that produces a graph in which every instruction points to every other instruction is considered *sound*. [2] While the ideal is somewhere in between, this is difficult to achieve with a scalable algorithm. Thus, different analyses require a different compromise between the two.

A further difficulty of control-flow graphs is accurately measuring *code coverage*, which is the measure of how much code is discovered by a control-flow graph. This is often complicated by the presence of *dead code*, code which is unreachable by any jumps.

B. Vulnerability Detection with Flow Modeling

Some vulnerabilities in a program can be discovered through the analysis of graphs of program properties.

Graph-based vulnerability discovery. Program property graphs (*e.g.*, control-flow graphs, data-flow graphs and control-dependence graphs) can be used to identify vulnerabilities in software. Initially applied to source code [60], [61], related techniques have since been extended to binaries [45]. These techniques rely on building a model of a bug, as represented by a set of nodes in a control-flow or data-dependency graph, and identifying occurrences of this model in applications. However, such techniques are geared toward searching for copies of vulnerable code, allowing the techniques to benefit from the pre-existing knowledge of an already existing vulnerability. Unlike these techniques, the focus of this paper is on the discovery of completely new vulnerabilities.

C. Vulnerability Detection with Data Modeling

Static analysis can also reason over abstractions of the data upon which an application operates.

Value-Set Analysis. One common static analysis approach is *Value-Set Analysis* (VSA) [6]. At a high level, VSA attempts to identify a tight over-approximation of the program state (*i.e.*, values in memory and registers) at any given point in the program. This can be used to understand the possible targets of indirect jumps or the possible targets of memory write operations. While these approximations suffer from a lack of accuracy, they are *sound*. That is, they may over-approximate, but never under-approximate.

By analyzing the approximated access patterns of memory reads and writes, the locations of variables and buffers can be identified in the binary. Once this is done, the recovered variable and buffer locations can be analyzed to find *overlapping* buffers. Such overlapping buffers can be, for example, caused by buffer overflow vulnerabilities, so each detection is one potential vulnerability.

IV. Background: Dynamic Vulnerability Discovery

Dynamic approaches are analyses that examine a program's execution, in an actual or emulated environment, as it acts given a specific input. In this section, we will focus specifically on dynamic techniques that are used for identifying vulnerabilities, rather than the general binary analysis techniques on which they are based.

Dynamic techniques are split into two main categories: concrete and symbolic execution. These techniques produce inputs that are highly *replayable*, but vary in terms of *semantic insight*.

A. Dynamic Concrete Execution

Dynamic concrete execution is the concept of executing a program in a minimally-instrumented environment. The program functions as normal, working on the same *domain* of data on which it would normally operate (i.e., ones and zeroes). These analyses typically reason at the level of single paths (i.e., "what path did the program take when given this specific input"). As such, dynamic concrete execution requires *test cases* to be provided by the user. This is a problem, as with a large or unknown dataset (such as ours) such test cases are not readily available.

1) Fuzzing: The most relevant application of dynamic concrete execution to vulnerability discovery is fuzzing. Fuzzing is a dynamic technique in which malformed input is provided to an application in an attempt to trigger a crash. Initially, such input was generated by hardcoded rules and provided to the application with little in-depth monitoring of the execution [38]. If the application crashed when given a specific input, the input was considered to have triggered a bug. Otherwise, the input would be further randomly mutated. Unfortunately, fuzzers suffer from the *test case* requirement. Without carefully crafted test cases to mutate, a fuzzer has trouble exercising anything but the most superficial functionality of a program.

Coverage-based fuzzing. The requirement for carefully-crafted test cases was partially mitigated with the advent of code-coverage-based fuzzing [39]. Code-coverage-based fuzzers attempt to produce inputs that maximize the amount of code executed in the target application based on the insight that the more code is executed, the higher the chance of executing vulnerable code. American Fuzzy Lop (AFL) [1], a

[2] Xu et. al. defines soundness and completeness of a CFG in the opposite way, where an empty graph is sound and a full graph is complete [59]. In this paper, we stick to the definition in Section III-A.

state-of-the art fuzzer responsible for the discovery of many recent vulnerabilities, uses a code coverage metric as its sole guiding principle, and its success at finding vulnerabilities has driven an increase of interest in fuzzing in recent years.

Coverage-based fuzzing suffers from a lack of semantic insight into the target application. This means that, while it is able to detect that a certain piece of code has not yet been executed, it cannot understand what parts of the input to mutate to cause the code to be executed.

Taint-based fuzzing. Another approach to improve fuzzing is the development of *taint-based* fuzzers [9], [62]. Such fuzzers analyze how an application processes input to understand what parts of the input to modify in future runs. Some of these fuzzers combine taint tracking with static techniques, such as data dependency recovery [30], [42]. Others introduce work from protocol analysis to improve fuzzing coverage [22].

While a taint-based fuzzer can understand what parts of the input should be mutated to drive execution down a given path in the program, it is still unaware of *how* to mutate this input.

2) Dynamic Symbolic Execution: Symbolic techniques bridge the gap between static and dynamic analysis and provide a solution to cope with the limited semantic insight of fuzzing. Dynamic symbolic execution, a subset of symbolic execution, is a dynamic technique in the sense that it executes a program in an emulated environment. However, this execution occurs in the *abstract* domain of *symbolic variables*. As these systems emulate the application, they track the state of registers and memory throughout program execution and the *constraints* on those variables. Whenever a conditional branch is reached, execution forks and follows *both* paths, saving the branch condition as a constraint on the path in which the branch was taken and the inverse of the branch condition as a constraint on the path in which the branch was not taken [49].

Unlike fuzzing, dynamic symbolic execution has an extremely high semantic insight into the target application: such techniques can reason about *how* to trigger specific desired program states by using the accumulated path constraints to retroactively produce a proper input to the application when one of the paths being executed has triggered a condition in which the analysis is interested. This makes it an extremely powerful tool in identifying bugs in software and, as a result, dynamic symbolic execution is a very active area of research.

Classical dynamic symbolic execution. Dynamic Symbolic Execution can be used directly to find vulnerabilities in software. Initially applied to the testing of source code [12], [13], dynamic symbolic execution was extended to binary code by Mayhem [16] and S2E [19]. These engines analyze an application by performing path exploration until a vulnerable state (for example, the instruction pointer is overwritten by input from the attacker) is identified.

However, the trade-offs discussed in Section II-A come into play: all currently proposed symbolic execution techniques suffer from very limited scalability due to the problem of *path explosion*: because new paths can be created at every branch, the number of paths in a program increases exponentially with the number of branch instructions in every path. There have been attempts to survive path explosion by *prioritizing* promising paths [11], [37] and by *merging* paths where the situation is appropriate [5], [35], [47]. However, in general, this challenge to pure dynamic symbolic execution analysis engines has not yet been surmounted, and (as we show later in this paper), most bugs discovered by such systems are *shallow*.

Symbolic-assisted fuzzing. One proposed way to address the path explosion problem is to offload much of the processing to faster techniques, such as fuzzing. This approach leverages the strength of fuzzing, *i.e.,* its speed, and attempts to mitigate the main weakness, *i.e.,* its lack of semantic insight into the application. Thus, researchers have paired fuzzing with symbolic execution [14], [15], [17], [28], [29], [54]. Such *symbolically-guided fuzzers* modify inputs identified by the fuzzing component by processing them in a dynamic symbolic execution engine. Dynamic symbolic execution uses a more in-depth understanding of the analyzed program to properly mutate inputs, providing additional test cases that trigger previously-unexplored code and allow the fuzzing component to continue making progress (i.e., in terms of code coverage).

Under-constrained symbolic execution. Another way to increase the tractability of dynamic symbolic execution is to execute only *parts* of an application. This approach, known as Under-constrained Symbolic Execution [26], [46], is effective at identifying *potential* bugs, with two drawbacks. First, it is not possible to ensure a proper context for the execution of parts of an application, which leads to many false positives among the results. Second, similar to static vulnerability discovery techniques, under-constrained symbolic execution gives up the replayability of the bugs it detects in exchange for scalability.

V. Background: Exploitation

Vulnerability discovery analyses actually discover *crashing inputs*. Triaging these crashing inputs – that is, understanding which crashes represent actual security issues, is outside of the scope of most such approaches. However, some work does exist on the reproduction and analysis of the discovered vulnerabilities. In this section, we go through the process of reproducing an identified crash, automatically generating the exploit to verify the security impact of the crash, and hardening the exploit to make it resilient in the presence of modern mitigation techniques.

A. Crash Reproduction

Most vulnerability discovery analyses execute a tested application in less-than-realistic conditions. For example, many fuzzers will *de-randomize* execution. That is, they will hard-code any sources of randomness, such as the PID of the executable, the current time, and so on. This is done for two main reasons. First, in most modern fuzzing approaches, there is an implicit assumption that the same input, provided to two instantiations of an application, will produce the same result both times. Second, the modeling of randomness in

other techniques, such as dynamic symbolic execution, is not a well-explored research area.

Because of de-randomization, the crashes reported by vulnerability discovery techniques might not be trivially replayable outside of the analysis environment. Consider the case of an application that generates a random token and requires the token to be provided by the user before entering an unsafe section of code and crashing. In the de-randomized analysis environment, the generated token will always have the same value, and the crashing input identified by the analysis will always take the same path, resulting in a crash. However, *outside* of the analysis environment, the token will always be different, and the previously-crashing input might instead take a *non-crashing* path.

Crashing inputs that are not trivially replayable generally fall into two categories.

Missing data. Vulnerability discovery techniques sometimes manage to "guess" correct response values without having first received them from the application. The token in our example is always a constant value in the de-randomized environment, and an analysis engine such as a fuzzer might accidentally guess it without first retrieving it from the program. When replaying the resulting crashing input outside of the analysis environment, the token value will not match and the crash will not occur.

Missing relationships. Techniques with low semantic insight, such as fuzzing, are unable to recover the *relationships* between data retrieved from the program and subsequent data provided to it. In our example, even though the crashing input might cause the application to provide the token to the user, so it can later be used to cause the crash, the output of the fuzzer lacks the relationship between the token value that the application provides to the user and the token value that the user must provide to the application.

In the case of missing data, the input is simply not replayable outside of the analysis environment, and a new crashing input might be found. Analyses exist that specialize in the identification of data leaks [42], but we have not yet implemented such analyses in `angr`.

In the latter case, the de-randomized crashing input must be converted into an *input specification* that defines how to communicate with an application in terms of the relationship between data received from the application and data later provided to it. One approach that does this is Replayer [43], which computes preconditions on program paths to understand how to reproduce a program path under real-world conditions.

B. Exploit Generation

With a productive vulnerability excavation engine utilizing one or more of the methods described above, many crashes might be produced for a tested application. However, not all of these crashes will be exploitable. An example of a non-exploitable input is a NULL-pointer dereference. Because modern operating systems disallow the mapping of memory at address 0, these previously-exploitable situations have been reduced to non-exploitable crashes. Understanding whether a crash is exploitable helps with the *triaging* of bugs (that is, understanding which bugs to investigate and fix first).

The obvious way to test if a crash would be exploitable is to try to exploit it. To this end, several systems have been proposed that attempt to take a crashing input and automatically convert it into an exploit for the application [4], [31], [51].

C. Exploit Hardening

In recent years, binary hardening techniques, such as non-executable stack regions and Address Space Layout Randomization (ASLR), have severely reduced the efficacy of traditional exploits, such as those generated by first-generation automatic exploitation engines. Thus, even an exploitable vulnerability might be mitigated by modern protections.

Current automatic exploitation techniques were designed before the widespread adoption of modern mitigation techniques, and modern software protections make the exploits they produce non-functional. To circumvent this, approaches have been created to automatically *harden* the exploits generated using current techniques against such defenses. Such techniques work by translating a traditional, shellcode-based exploit into an equivalent exploit utilizing Return-Oriented Programming [52]. As such, an automatic approach to constructing Return-Oriented Programs is required, and several such approaches have been developed [18], [48].

VI. Analysis Engine

The analyses that we described in sections III, IV and V were proposed at various times over the last several years, implemented with different technologies, and evaluated on disparate datasets with varying methodologies. This is problematic, as it makes it hard to understand the relative effectiveness of different approaches and their applicability to different types of applications.

To alleviate this problem, we have developed a flexible, capable, next-generation binary analysis system, `angr`, and used it to implement a selection of the analyses we presented in the previous sections. This section describes the analysis system, our design goals for it, and the impact that this design has had on the analysis of realistic binaries.

A. Design Goals

Our design goals for `angr` are the following:

Cross-architecture support. With the rise of embedded devices, often running ARM and MIPS processors, modern software is made for varying hardware architectures. This is a departure from the previous decade, where x86 support was enough for most analysis engines: a modern binary analysis engine must be able to perform cross-architecture analyses. Furthermore, 32-bit processors are no longer the standard; a modern analysis engine must support analysis of 64-bit binaries.

Cross-platform support. In a similar vein to cross-architecture support, a modern analysis system must be able to analyze software from different operating systems. This means that concepts specific to individual operating

systems must be abstracted, and support for *loading* different executable formats must be implemented.

Support for different analysis paradigms. A useful analysis engine must provide support for the wide range of analyses described in earlier sections. This requires that the engine itself abstract away, and provide different types of memory models as well as data domains.

Usability. The purpose of `angr` is to provide a tool for the security community that will be useful in reproducing, improving, and creating binary analysis techniques. As such, we strove to keep `angr`'s learning curve low and its usability high. `angr` is almost completely implemented in Python, with a concise, simple API that is easily usable from the IPython interactive shell [44]. While Python results in constant-time lower performance than other potential language choices, most binary analysis techniques suffer from *algorithmic* slowness, and the language-imposed performance impact is rarely felt. When language overhead *is* important, `angr` can run in the Python JIT engine, PyPy for a significant speed increase.

Our goal was for `angr` to allow for the reproduction of a typical binary analysis technique, on top of our platform, in about a week. In fact, we were able to reproduce Veritesting [5] in eight days, guided symbolic execution in a month, AEG [4] in a weekend, Q [48] in about three weeks, and under-constrained symbolic execution [46] in two days. It is hard to produce an implementation effort estimate for dynamic symbolic execution and value-set analysis, as we implemented those while building the system itself over two years.

In order to meet these design goals, we had to carefully build our analysis engine. We did this by creating a set of modular building blocks for various analyses, being careful to maintain strict separation between them to reduce the number of assumptions that higher-level parts of `angr` (such as the state representation) make about the lower-level parts (such as the data model). This makes it easier for us to mix and convert between analyses on-the-fly. We hope that it will also make it easier for other researchers to reuse individual modules of `angr`. In the next several sections, we discuss the technical design of each `angr` submodule.

B. Submodule: Intermediate Representation

In order to support multiple architectures, we translate architecture-specific native binary code into an intermediate representation (IR) atop which we implement the analyses. Rather than writing our own "IR lifter", which is an extremely time-consuming engineering effort, we leveraged libVEX, the IR lifter of the Valgrind project. libVEX produces an IR, called VEX, that is specifically designed for program analysis. We used PyVEX, which we originally wrote for Firmalice [53], to expose the VEX IR to Python. By leveraging VEX, we can provide analysis support for 32-bit and 64-bit versions of ARM, MIPS, PPC, and x86 (with the 64-bit version of the latter being amd64) processors. Improvements are constantly being made by Valgrind contributors, with, for example, a port to the SPARC architecture currently underway.

As we will discuss later, there is no fundamental restriction for `angr` to always use VEX as its IR. As implemented, supporting a different intermediate representation would be a straightforward engineering effort.

C. Submodule: Binary Loading

The task of loading an application binary into the analysis system is handled by a module called `CLE`, a recursive acronym for `CLE Loads Everything`. `CLE` abstracts over different binary formats to handle loading a given binary and any libraries that it depends on, resolving dynamic symbols, performing relocations, and properly initializing the program state. Through `CLE`, `angr` supports binaries from most POSIX-compliant systems (Linux, FreeBSD, etc.), Windows, and the DECREE OS created for the DARPA Cyber Grand Challenge.

`CLE` provides an extensible interface to a binary loader by providing a number of base classes representing binary objects (i.e., an application binary, a POSIX `.so`, or a Windows `.dll`), segments and sections in those objects, and symbols representing locations inside those sections. `CLE` uses file format parsing libraries (specifically, `elftools` for Linux binaries and `pefile` for Windows binaries) to parse the objects themselves, then performs the necessary relocations to expose the memory image of the *loaded* application.

D. Submodule: Program State Representation/Modification

The `SimuVEX` module is responsible for representing the *program state* (that is, a snapshot of values in registers and memory, open files, *etc.*). The state, named `SimState` in `SimuVEX` terms, is implemented as a collection of *state plugins*, which are controlled by *state options* specified by the user or analysis when the state is created. Currently, the following state plugins exist:

Registers. `SimuVEX` tracks the values of registers at any given point in the program as a state plugin of the corresponding program state.

Symbolic memory. To enable symbolic execution, `SimuVEX` provides a symbolic memory model as a state plugin. This implements the indexed memory model proposed by Mayhem [16].

Abstract memory. The abstract memory state plugin is used by static analyses to model memory. Unlike symbolic memory, which implements a continuous indexed memory model, the abstract memory provides a *region*-based memory model which is used by most static analyses.

POSIX. When analyzing binaries for POSIX-compliant environments, `SimuVEX` tracks the *system state* in this state plugins. This includes, for example, the files that are open in the symbolic state. Each file is represented as a memory region and a symbolic position index.

Log. `SimuVEX` tracks a log of everything that is done to the state (i.e., memory writes, file reads, etc.) in this plugin.

Inspection. `SimuVEX` provides a powerful debugging interface, allowing breakpoints to be set on complex conditions, including taint, exact expression makeup, and

symbolic conditions. This interface can also be used to *change* the behavior of SimuVEX. For example, memory reads can be instrumented to emulate memory-mapped I/O devices.

Solver. The Solver is a plugin that exposes an interface to different data domains, through the data model provider (Claripy, discussed below). For example, when this plugin is configured to be in symbolic mode, it interprets data in registers, memory, and files symbolically and tracks path constraints as the application is analyzed.

Architecture. The architecture plugin provides architecture-specific information that is useful to the analysis (i.e., the name of the stack pointer, the wordsize of the architecture, etc). The information in this plugin is sourced from the archinfo module, that is also distributed as part of angr.

These state plugins provide building blocks that can be combined in various ways to support different analyses.

Additionally, SimuVEX implements the base unit of an analysis: representing the semantic changes made to a program state by a block of application code (in SimuVEX terminology, such a block of code is called a SimRun). That is, SimuVEX provides the capability to process an *input* state through a block of VEX-represented code, and to generate an *output* state (or a set of output states, in case we encounter a block from which multiple output states are possible, such as a conditional jump). Again, this part of SimuVEX is modular: in addition to VEX translations of basic blocks, SimuVEX currently allows the user to provide a handcrafted Python function as a SimRun, providing a powerful way to instrument blocks with Python code. In fact, this is how we implement our environment model: system calls are implemented as Python functions that modify the program state.

E. Submodule: Data Model

The *values* stored in the registers and memory of a SimState are represented by abstractions provided by another module, Claripy.

Claripy abstracts all values to an internal representation of an *expression* that tracks all operations in which it is used. That is, the expression x, added to the expression 5, would become the expression $x + 5$, maintaining a link to x and 5 as its arguments. These expressions are represented as "expression trees" with values being the leaf nodes and operations being non-leaf nodes.

At any point, an expression can be translated into data domains provided by Claripy's *backends*. Specifically, Claripy provides backends that support the concrete domain (integers and floating-point numbers), the symbolic domain (symbolic integers and symbolic floating point numbers, as provided by the Z3 SMT solver [25]), and the value-set abstract domain for Value Set Analysis [6]. Claripy is easily extensible to other backends. Specifically, implementing other SMT solvers would be interesting, as work has shown that different solvers excel at solving different types of constraints [8].

User-facing operations, such as interpreting the constructs provided by the backends (for example, the symbolic expression $x+1$ provided by the Z3 backend) into Python primitives (such as possible integer solutions for $x + 1$ as a result of a constraint solve) are provided by *frontends*. A frontend augments a backend with additional functionality of varying complexity. Claripy currently provides several frontends:

FullFrontend. This frontend exposes symbolic solving to the user, tracking constraints, using the Z3 backend to solve them, and caching the results.

CompositeFrontend. As suggested by KLEE and Mayhem, splitting constraints into independent sets reduces the load on the solver. The CompositeFrontend provides a transparent interface to this functionality.

LightFrontend. This frontend does not support constraint tracking, and simply uses the VSA backend to interpret expressions in the VSA domain.

ReplacementFrontend. The ReplacementFrontend expands the LightFrontend to add support for *constraints* on VSA values. When a constraint (i.e., $x+1 < 10$) is introduced, the ReplacementFrontend analyzes it to identify bounds on the variables involved (i.e., $0 <= x <= 8$). When the ReplacementFrontend is subsequently consulted for possible values of the variable x, it will intersect the variable with the previously-determined range, providing a more accurate result than VSA would otherwise be able to produce.

HybridFrontend. The HybridFrontend combines the FullFrontend and the ReplacementFrontend to provide fast approximation support for symbolic constraint solving. While Mayhem [16] hinted at such capability, to our knowledge, angr is the first publicly available tool to provide this capability to the research community.

This modular design allows Claripy to combine the functionalities provided by the various data domains in powerful ways and to expose it to the rest of angr.

F. Submodule: Full-Program Analysis

The analyst-facing part of angr provides complete analyses, such as dynamic symbolic execution and control-flow graph recovery. The "entry point" into these analyses is the Project, representing a binary with its associated libraries. From this object, all of the functionality of the other submodules can be accessed (i.e., creating states, examining shared objects, retrieving intermediate representation of basic blocks, hooking binary code with Python functions, etc.). Additionally, there are two main interfaces for full-program analysis: Path Groups and Analyses.

Path Groups. A PathGroup is an interface to dynamic symbolic execution – it tracks paths as they run through an application, split, or terminate.The creation of this interface stemmed from frustration with the management of paths during symbolic execution. Early in angr's development, we would implement ad-hoc management of paths for each analysis that would use symbolic

execution. We found ourselves re-implementing the same functionality: tracking the hierarchy of paths as they split and merge, analyzing which paths are interesting and should be prioritized in the exploration, and understanding which paths are not promising and should be terminated. We unified the common actions taken on groups of paths, creating the `PathGroup` interface.

Analyses. `angr` provides an abstraction for any full program analysis with the `Analysis` class. This class manages the lifecycle of static analyses, such as control-flow graph recovery, and complex dynamic analyses as those presented in Section IX.

When `angr` identifies some *truth* about a binary (i.e., "the basic block at address X can jump to the basic block at address Y"), it stores it in the *knowledge base* of the corresponding Project. This shared knowledge base allows analyses to collaboratively discover information about the application.

G. Open-Source Release

We started to work on `angr` with the goal of developing a platform on which we could implement new binary analysis approaches. As we faced the unexpected challenges associated with the analysis of realistic binaries, we realized that such an analysis engine would be extremely useful to the security community. We have open-sourced `angr` in the hope that it will provide a basis for the future of binary analysis, and it will free researchers from the burden of having to re-address the same challenges over and over. `angr` is implemented in just over $65,000$ lines of code, usable directly from the IPython shell or as a python module, and easily installable via the standard Python package manager, `pip`.

The open-source release of `angr` includes the analysis engine modules (as described in Sections VI-A through VI-F) on top of which we implemented the applications discussed in Section XV. Of the latter, we have open-sourced our control-flow graph recovery, the static analysis framework, our dynamic symbolic execution engine, and the under-constrained symbolic execution implementation. While we plan to release the other applications in the future, they are currently in a state that is a mix of being prototype-level code and being actively applied toward the DARPA Cyber Grand Challenge.

`angr` has been met with extreme enthusiasm by the community. In the first 3 months after the open-source release, we gathered almost 500 "stars" (measures of persons valuing the software) on GitHub across the different modules that make up the system. In the same time period, `angr` had roughly 6,000 total installations via `pip` and an average of 20 "clones" of the Git repository weekly. `angr` has already been used in at least one class project in another institution to introduce students to binary analysis. Additionally, we are aware of several other institutions using it as a basis to build research prototypes and a number of corporations evaluating it for usability in commercial binary analysis systems.

VII. Implementation: CFG Recovery

We will describe the process that `angr` uses to generate a CFG, including specific techniques that were developed to improve the completeness and soundness of the final result.

Given a specific program, `angr` performs an iterative CFG recovery, starting from the entry point of the program, with some necessary optimizations. `angr` leverages a combination of *forced execution*, *backwards slicing*, and *symbolic execution* to recover, where possible, all jump targets of each indirect jump. Moreover, it generates and stores a large quantity of data about the target application, which can be used later in other analyses such as data-dependence tracking.

This algorithm has three main drawbacks: it is slow, it does not automatically handle "dead code", and it may miss code that is only reachable through unrecovered indirect jumps. To address this issue, we created a secondary algorithm that uses a quick disassembly of the binary (without executing any basic block), followed by heuristics to identify functions, intra-function control flow, and direct inter-function control flow transitions. The secondary algorithm, however, is much less accurate – it lacks information about reachability between functions, is not context sensitive, and is unable to recover complex indirect jumps.

In the reminder of this section, we discuss our advanced recovery algorithm, which we dub `CFGAccurate`. We then discuss our fast algorithm, `CFGFast`, in Section VII-F.

A. Assumptions

`angr`'s `CFGAccurate` makes several assumptions about binaries to optimize the run time of the algorithm.

1) All code in the program can be distributed into different functions.
2) All functions are either called by an explicit call instruction (or its equivalents), or are preceded by a tail jump (an optimization, often used to reduce stack space for recursive functions, in which a call at the very end of a function is changed to a *jump* so that the newly called function simply reuses the return address of its caller) in the control flow.
3) The stack cleanup behavior of each function is predictable, regardless of where it is called from. This lets `CFGAccurate` safely skip functions that it has already analyzed while analyzing a caller function and keep the stack balanced.

These assumptions place constraints on the types of binaries that `angr` is designed to analyze. Assumptions 1, 2, and 3 require that the binary being analyzed is not obfuscated and behaves in a "normal" way. We *can* remove those assumptions when analyzing obfuscated or abnormal binaries, but this would lead to a higher run time of the CFG recovery.

Our CFG recovery code is built upon techniques proposed by related literature [21], [34], [50], [58], [59]. However, these techniques make assumptions that are overly strict or are unrealistic for real-world binaries. Specifically, we do *not* assume any of the following, unlike the work that our CFG recovery is based on:

1) All functions return to the next instruction after their call-site [59].

2) The jump target of an indirect branch is always determined by a control flow path, not by a program state or context [59]. For example, some existing literature assumes that indirect jumps are all *computed*, as opposed to being passed in as a function pointer from prior contexts.
3) Expressions for jump targets of indirect jumps must match a set of common idioms [21], [58]. Unlike existing work, we make no assumptions on the type of operations that can be applied to pointers.
4) The stack pointer is the same before entering a function as it is after returning from it.
5) No two functions overlap (in other words, they cannot share basic blocks [34].) `CFGAccurate` handles functions that share code.
6) Additional information, such as symbol tables or relocation information, is available [50].

The actual algorithm to recover a control-flow graph from a binary is described in the next few sections.

B. Iterative CFG Generation

Unfortunately, no single technique meets `CFGAccurate`'s goal of recovering a complete and sound CFG. Thus, `CFGAccurate` constructs a CFG by interleaving a series of techniques to achieve speed and completeness. Specifically, four techniques are used: forced execution, lightweight backward slicing, symbolic execution, and value set analysis. The CFG to be iteratively recovered by these techniques, C, is initialized with the basic block at the entry point of the application.

Throughout CFG recovery, `CFGAccurate` maintains a list of indirect jumps, L_j, whose jump targets have not been resolved. When the analysis identifies such a jump, it is added to L_j. After each iterative technique terminates, `CFGAccurate` triggers the next one in the list. This next technique may resolve jumps in L_j, may add new unresolved jumps to L_j, and may add basic blocks and edges to the CFG C. `CFGAccurate` terminates when a run of all techniques results in no change to L_j or C, as that means that no further indirect jumps can be resolved with any available analysis.

C. Forced Execution

`angr`'s `CFGAccurate` leverages the concept of Dynamic Forced Execution for the first stage of CFG recovery [59]. Forced Execution ensures that both directions of a conditional branch will be executed at every branch point.

`CFGAccurate` maintains a work-list of basic blocks, B_w, and a list of analyzed blocks, B_a. When the analysis starts, it initializes its work-list with all the basic blocks that are in C but not in B_a. Whenever `CFGAccurate` analyzes a basic block from this work-list, the basic block and any *direct* jumps from the block are added to C. Indirect jumps, however, cannot be handled this way. Under forced execution, the targets of indirect jumps may differ from those of an actual run of the program because forced execution will execute code in an unexpected order. Thus, each indirect jump is stored in the list L_j for later analysis.

As it cannot resolve any indirect jumps, this analysis functions as a fast-pass CFG recovery analysis to quickly seeds the other analyses with detected basic blocks and unresolved indirect jumps.

D. Symbolic Execution

The main issue with dynamic forced execution is the presence of indirect jumps, as there is no way to make sure that the target of an indirect jump is correctly resolved. On the one hand, an indirect jump may be completely unresolvable (i.e., the forced execution resulted in a state where the jump target is read from uninitialized memory), which leaves a broken control flow transition in the recovered CFG. On the other hand, an indirect jump may also be partially solvable (i.e. our analysis only retrieves a portion of all the possible jump targets).

For each jump $J \in L_j$, `CFGAccurate` traverses the CFG backwards until it find the first *merge point* (that is, multiple paths converging on the way to the indirect jump) or up to a threshold number of blocks (empirically, we found a reasonable threshold to be 8). From there, it performs forward symbolic execution to the indirect jump and uses a constraint solver to retrieve possible values for the target of the indirect jump.

`CFGAccurate` considers a jump successfully resolved if the computed set of possible targets is smaller than a threshold size. We use a value of 256 for this threshold but we have found that, in practice, in the cases where jumps are *not* resolved successfully, this value is *unconstrained* (meaning, the set of possible targets is bounded only by the number of bits in the address).

If the jump is resolved successfully, J is removed from L_j and edges and nodes are added to the CFG for each possible value of the jump target.

E. Backward Slicing

`angr`'s forced execution and symbolic execution analyses fail to resolve many of the unresolved jumps due to the lack of *context*. Those analyses are carried out in a context-insensitive manner: if a function takes pointer as an argument, and that pointer is used as the target of an indirect jump, the analyses will be unable to resolve it.

To achieve better completeness, our CFG generation requires a context-sensitive component. We accomplish this with *backward slicing*. `CFGAccurate` computes a backward slice starting from the unresolved jump. The slice is extended through the beginning of the previous *call context*. That is, if the indirect jump being analyzed is in a function F_a that is called from both F_b and F_c, the slice will extend backward from the jump in F_a and contain two start nodes: the basic block at the start of F_b and the one at the start of F_c.

`CFGAccurate` then executes this slice using `angr`'s symbolic execution engine and uses the constraint engine to identify possible targets of the symbolic jumps, with the same threshold of 256 for the size of the solution set for the jump target. If the jump target is resolved successfully, the jump is removed from L_j and the edge representing the control

flow transition, and the target basic blocks are added to the recovered CFG.

F. CFGFast

The goal of the fast CFG generation algorithm is to generate a graph, with high code coverage, that identifies at least the location and content of functions in the binary. This graph lacks much of the *control flow*, so it is not complete. However, such a graph can still be useful for both manual and automated analysis of binaries.

`CFGFast` carries out the following steps:

Function identification. We use hard-coded function prologue signatures, which can be generated from techniques like ByteWeight [7], to identify functions inside the application. If the application includes *symbols*, specifying the locations of functions, they are also used to seed the graph with function start positions. Additionally, the basic block representing the entry point of the program is added to the graph.

Recursive disassembly. Recursive disassembly is used to recover the direct jumps within the identified functions.

Indirect jump resolution. Lightweight alias analysis, data-flow tracking, combined with pre-defined strategies are used to resolve intra-function control flow transfers. Currently `CFGFast` includes strategies for jump table identification and indirect call target resolution.

The goal is to quickly recover a CFG with a high coverage, without a concern for understanding the reachability of functions from one another.

G. Using the CFG Recovery

`angr` exposes the CFG recovery algorithms as two analyses: `CFGFast` and `CFGAccurate`. These analyses output CFG data to `angr`'s knowledge base, as discussed in Section VI-F. This data can then be used in the course of manual analysis or later automated analyses.

VIII. Implementation: Value Set Analysis

Once a CFG is generated, more advanced analyses can be run. One of these is Value-Set Analysis [6]. Value-Set Analysis (VSA) is a static analysis technique that combines numeric analysis and pointer analysis for binary programs. It uses an abstract domain, called the Value-Set Abstract domain, for approximating possible values that registers or abstract locations may hold at each program point.

VSA analyzes a program until it reaches a *fix-point* for all program points in the function. This fix-point represents a tight over-approximation of all values that any register or abstract memory location can have at any point in the function. With respect to, for example, a memory write to a computed address A, consulting the values of A in the computed fix-point will contain a complete list of all possible write targets.

The original VSA design, proposed by Balakrishnan et al. [6], does not perform well when analyzing real-world binaries. To make VSA work on such binaries, we had to develop a number of improvements to increase the precision of our analysis.

Creating a discrete set of strided-intervals. The basic data type of VSA, the strided interval, is essentially an approximation of a set of numbers. It is great for approximating a set of normal concrete values. However, if those values are used as jump targets in the program, the over-approximating nature of strided-intervals yields unsoundness in our recovered CFG by creating control flow transitions to addresses that should not be jump targets. To effectively solve this problem, we developed a new data type called "strided interval set", which represents a set of strided intervals that are not unioned together. A strided interval set will be unioned into a single strided interval only when it contains more than K elements, where K is a threshold that can be adjusted. In our model discussed in Section II-A, this threshold controls a trade-off of semantic insight versus scalability – a higher value of K allows us to maintain high precision, but comes at a cost of increased analysis complexity.

Applying an algebraic solver to path predicates. Tracking branch conditions helps us constrain variables in a state after taking a conditional exit or during a merging procedure, which produces a more precise analysis result. Affine-Relation Analysis has been proposed as a technique to track these conditions [40]. However, it is both complicated to implement (generally leading to support for very few arithmetic operations in constraint expressions), and is computationally expensive in reality. Our solution is to implement a lightweight algebraic solver that works on the strided interval domain, based on modulo arithmetic which take care of some of the affine relations. When a new path predicate is seen (i.e., when following a conditional branch), we attempt to simplify and solve it to obtain a number range for the variables involved in the path predicate. Then we perform an intersection between the newly generated number range and the original values for each corresponding variable. This allows us to continuously refine the result of our value-set analysis as new branch conditions are encountered, increasing the precision of the eventual fix-point.

Adopting a signedness-agnostic domain. As originally proposed, VSA operates on a signed strided interval domain, which assumes all values are signed. That is, for an n-bit strided-interval with l as its lower bound and h as its upper bound, we always have $l \in [-2^{n-1}, 2^{n-1}-1] \wedge h \in [-2^{n-1}, 2^{n-1}-1] \wedge l \leq h$. However, this results in heavily over-approximated results of unsigned arithmetic calculations. In fact, this over-approximation is exacerbated by the fact that, since jump addresses are unsigned, the computation of jump addresses generally relies on unsigned values (i.e., in the case of unsigned comparisons). The solution to this problem is to adopt a signedness-agnostic domain for our analysis. *Wrapped Interval Analysis* [41] is such

an interval domain for analyzing LLVM code, which takes care of signed and unsigned numbers at the same time. We based our signedness-agnostic strided-interval domain on this theory, applied to the VSA domain.

We use VSA for memory corruption detection in three phases. First, we collect all read and write access patterns in the program during the VSA. On top of those access patterns, we perform a variable recovery for variables on both the stack and heap regions. Our implementation is similar to the variable recovery in TIE [36]. Next, we scan all stack and heap regions to find abnormal buffers, including a) overlapping buffers, and b) out-of-bound buffers. Then we simply report all abnormal buffers as potential memory corruptions.

A. Using VSA

The main interface that `angr` provides into a full-program VSA analysis is the *Value Flow Graph*. The VFG is an enhanced CFG that includes the *program state* representing the VSA fix-point at each program location. Depending on the parameters passed to the VFG analysis, this can include a single function, a tree of function calls, or the entire program.

The program states contained in the VFG present memory in an abstract layout provided by `SimuVEX` (specifically, the `SimAbstractMemory` memory model), with values in memory represented by value-sets, as provided by `Claripy`. We performed our buffer overlap analysis over the data contained in these program states by analyzing the range of values that memory accesses may take.

IX. Implementation: Dynamic Symbolic Execution

The dynamic symbolic execution module of our analysis platform is mainly based on the techniques described in Mayhem [16]. Our implementation follows the same memory model and path prioritization techniques. This module represents one of the core functionalities of `angr`, other analyses, such as Veritesting and under-constrained symbolic execution, use it as a base.

We use `Claripy`'s interface into Z3 to populate the symbolic memory model (specifically, `SimSymbolicMemory`) provided by `SimuVEX`. Individual execution paths through a program are managed by `Path` objects, provided by `angr`, which track the actions taken by paths, the path predicates, and various other path-specific information. Groups of these paths are managed by `angr`'s `PathGroup` functionality, which provides an interface for managing the splitting, merging, and filtering of paths during dynamic symbolic execution.

`angr` has built-in support for Veritesting [5], implementing it as a `Veritesting` analysis and exposing transparent support for it with an option passed to `PathGroup` objects. This advanced state merging technique helps mitigate the problem of exponential state explosion by statically (and selectively) merging paths.

X. Implementation: Under-Constrained Symbolic Execution

We implemented under-constrained symbolic execution (UCSE), as proposed in UC-KLEE [46], and dubbed it `UC-angr`. UCSE is a dynamic symbolic execution technique where execution is performed on each function separately. Since the analysis cannot reason about *how* to get to the specific function, detections by UCSE are not replayable. Because each function is generated without its *context* (i.e., the arguments and global variables with which it is called in actual executions), the analysis is not accurate and suffers from false positives.

UCSE tags missing context in the state as *under-constrained*. When such under-constrained data is used as a pointer, a new under-constrained region is created and the pointer is directed at the new region. This "on-demand" memory allocation enables code that manages complex data structures to be analyzed. When a security violation is identified (i.e., a write to the saved return address on the stack), the values involved are checked for their *under-constrained* status. Under certain conditions (i.e., if all data involved is under-constrained), the violation is filtered out as a false positive.

We made two changes to the technique described in UCSE:

Global memory under-constraining. The original UC-KLEE implementation does not treat access to global memory as under-constrained. However, such memory is part of the program context that is impossible to predict with UCSE, since, when analyzing a given function, global data could have already potentially been overwritten. Thus, we mark all global data as under-constrained, allowing us to lower our false positive rate.

Path limiters. The original UC-KLEE implementation had several built-in limitations to prevent a path explosion. For example, they would limit the depth of under-constrained pointer dereferences to avoid a search through an under-constrained linked list never terminating. We added an additional limiter: we abort the analysis of a function when we find that it is responsible for a path explosion. We detect this by hard-coding a limit (in our experiments, we used an empirically-determined limit of 64 paths) and, when a single function branches over this many paths, we replace the function with an immediate `return`, and rewind the analysis from the call site of that function. This keeps the analysis tractable by avoiding path explosions, but makes the analysis even less accurate.

False positive filtering. We introduced several additional false positive filters into our implementation of UC-`angr`. Specifically, when we detect an exploitable state, we attempt to ensure that the state is not incorrectly made exploitable by a lack of constraints on under-constrained data. First, we perform a constraint solve with an additional constraint, E, that expresses the fact that the state is *not* exploitable (i.e., if the security violation was an overwrite of the return address, we constrain the state so that the return address could *not* have been overwritten). Then, we constrain each under-constrained value to its possible solution from this unexploitable state. We call these constraints U. Finally, we remove the constraint E, keeping the constraints U, and check

that the state can still be exploited. If it can, this means that the function likely has some inherent flaw, and the flaw does not necessarily depend on missing *data* from the context. Note that the flaw could still be a false positive due to missing *constraints*, or due to the limited context on data that is not under-constrained.

`UC-angr` is implemented as a `SimState` plugin that tracks under-constrained data accesses and carries out the required relocations. Once this plugin is initialized, under-constrained symbolic execution can be performed using the same `PathGroup` paradigm as dynamic symbolic execution.

XI. Implementation: Symbolic-Assisted Fuzzing

While we give a summary of our symbolic-assisted fuzzing implementation here, the full approach, called `Driller`, is detailed in a separate paper [54].

Our implementation of symbolic-assisted fuzzing uses the AFL fuzzer as its foundation and `angr` as its symbolic tracer. By monitoring AFL's performance, we can decide when to begin symbolically-tracing the inputs that AFL has created. To make this decision, we act on the rate at which the fuzzer is discovering new state transitions. If AFL reports that it has discovered no new state-transitions after performing a round of mutations of the input, we assume the fuzzer to be having trouble making progress, and invoke `angr` on all paths AFL has deemed as *unique* (i.e., any path that has a jump, identified by a tuple of the source and destination address, that no other path has), looking for transitions that AFL was unable to find inputs for.

`Driller`'s symbolic component is implemented using `angr`'s symbolic execution engine, so as to symbolically trace paths based on the concrete inputs provided by AFL. This avoids the path explosion problem inherent to symbolic execution, as each concrete input corresponds to a single (traced) path, and these inputs are heavily filtered by AFL to ensure that only promising ones are traced. Each concrete input corresponds to an individual path in a `PathGroup`. At each step of the `PathGroup`, every branch is checked so as to ensure that the most recent jump instruction leads to a path previously unknown to AFL. When such a jump is found, the SMT solver is queried to create an input that would drive execution to the new jump. This input is fed back to AFL, which goes on to mutate it in future fuzzing steps. This feedback loop allows us to balance expensive symbolic execution time with cheap fuzzing time, and mitigates fuzzing's low semantic insight into program operation.

XII. Implementation: Crash Reproduction

We implemented the approach proposed by Replayer [43] to recover missing relationships between input values (i.e., values that the attacker sends) and output values (i.e., values that the attacker leaks from the application).

Our implementation of Replayer is built atop our symbolic execution engine. We can define the problem of replaying a crashing input as the search for an input specification i_s to bring a program from an initial state s to a crash state q. Our algorithm takes, as input, the program P, an initial state s_a (i.e., the state at the entry point of the executable), the crash state q_a, and the input i_a that brings s_a to q_a in the instrumented (de-randomized) environment, but does not properly replay in an uninstrumented environment. Our implementation symbolically executes the path from s_a to q_a, using the input i_a. It records all constraints that are generated while executing P. Given the constraints, the execution path, the program P, and the new initial state s_b, we can symbolically execute P with an *unconstrained* symbolic input, following the previously recorded execution path until the new crash state q_b is reached. At this point, the input constraints on the input and output can be analyzed, and relationships between them can be recovered. This relationship data is used to generate the input specification i_s, allowing the crashing input to be replayed.

The implementation proposed by Replayer has two main limitations in its application to crash reproduction. First, as we discuss in Section V-A, it is possible that a given crash does not retrieve all of the data that is required to properly replay the crash. Replayer is unable to handle these cases, and new crashing inputs must be found.

Second, Replayer uses only the exact path, as executed by the application in the de-randomized environment while processing the crashing input, to generate the input specification. If the execution trace of a binary changes, based on the exact value of random data, then Replayer cannot compute the correct input. For example, if the random cookie introduces path predicates, by causing the execution of a specific path through a decoding function, replaying execution with that exact path will constrain the cookie to a value that might differ from the initial one. When this happens, the replayed cookie will not be correct, and the replaying attempt will fail. As we will discuss later, AEG is facing a similar limitation. This suggests that research in this area could make progress for both of these tasks.

XIII. Implementation: Exploit Generation

By implementing algorithms similar to those described in AXGEN [51], AEG [4] and Mayhem [16], we were able to evaluate the effectiveness of the current state of the art in automatic exploit generation. Our implementation allows us to create exploits for vulnerabilities, allowing the attacker to take control of the program's execution by overwriting a saved instruction pointer (*e.g.,* by overwriting function pointers, or exploiting buffer overflows on the stack).

Vulnerable States. Unlike AEG/Mayhem, but similar to AXGEN, we generate exploits by performing concolic execution on crashing program inputs using `angr`. We drive concolic execution forward, forcing it to follow the same path as a dynamic trace gathered by concretely executing the crashing input applied to the program. Concolic execution is stopped at the point where the program crashed, and we inspect the symbolic state to determine the cause of the crash and measure exploitability. By counting the number of symbolic bits in certain registers, we can triage a crash into a number of categories such as `frame pointer`

`overwrite`, `instruction pointer overwrite`, or `arbitrary write`, among others.

Instruction Pointer Overwrite Technique. The simplest exploitable bug we can encounter is where symbolic bits appear in the instruction pointer at crash time. When detecting that symbolic bits are contained in the instruction pointer, we can constrain our instruction pointer to point to either a controlled sequence of instructions, such as shellcode, or a ROP gadget that pivots the stack to a symbolic buffer where we can execute a ROP chain (generated by our exploit hardening step). `angr` itself handles many of the implementation details discussed in AEG and AXGEN, such as taint tracking and path condition building, allowing us to limit ourselves to finding symbolic memory buffers and applying constraints to register values to generate an exploit, as proposed by these approaches.

Exploiting CGC Binaries. The Cyber Grand Challenge hosts the game on a custom OS which includes only 7 system calls. The lack of system calls which can execute programs and open files means exploitation within the Cyber Grand Challenge is limited to demonstrating register control and the ability to read and write memory. By DARPA standards, two type of exploits exist for the CGC:

- A Type 1 exploit demonstrates that the attacker controls a general purpose register and the instruction pointer register.
- A Type 2 exploit demonstrates that the attacker can perform a controlled read from the process memory space.

Out of the 126 binaries we applied AEG to, we succeeded in exploiting only a total of 4 binaries. For only two of these binaries, we were able to generate a "Type 2" exploit. Both of these "Type 2" exploits were unable to be hardened with ROP and resorted to jumping to shellcode. Additionally, AEG was only able to generate 2 hardened, ROP "Type 1" exploits. We believe these results show that much more work in the field of automated exploit generation is to be done, and that the current methods are not well-applicable to modern vulnerabilities.

Challenges Faced. Here we demonstrate some of the challenges that our tool faced while attempting to exploit Cyber Grand Challenge binaries, using `CROMU00019` [24]. We will focus on the exploitation of the second vulnerability mentioned in this challenge's `README` (specifically, a buffer overflow on the stack that exists during the decoding of an attacker-supplied string).

The major issue we ran into during exploit generation was the presence of path predicates that constrained each byte of the overflowing data to being a single value, despite the values of these bytes being chosen based on symbolic input. `CROMU00019` demonstrates this in its `decode` function. Each byte of the payload takes a branch of the switch statement contained in `decode`, placing restrictive predicates on our path representing the vulnerable state. While the arms of this switch statement are taken based on symbolic data, the data returned is concrete, and each of these arms represents a separate path through the program. The traditional AEG approach assumes the ability to place the proper constraints on symbolic data to carry out control flow hijacking, but this behavior requires finding the *single* path through the `decode` function which places our desired bytes into the output buffer.

The solution to this problem would be to search for a single path which performs a desirable control flow hijack out of the many paths which present vulnerable conditions. However, modern exploit generation capabilities do not have this capacity, and cases like these prevent many of the stack buffer overflow vulnerabilities presented in the CGC Qualifier event from being exploited with the current state-of-the-art automatic exploit generation.

XIV. Implementation: Exploit Hardening

To harden exploits against modern mitigation techniques, we implemented a ROP chain compiler based on the ideas in Q [48]. This means that we can automatically generate ROP payloads to fulfill an end goal, such as writing data to memory or calling an arbitrary function in a library. This section focuses on the differences and improvements we made over Q itself.

Our approach comprises the following steps:

Gadget discovery. We scan all executable code in the application, at every byte offset, to identify ROP gadgets and classify them according to their effects. For example, the instruction sequence: `mov [ebx], eax; pop ebx; ret` would be classified as a memory write and a register load. To carry out the classification, our analysis leverages the action history provided by `angr`'s `Path` objects and symbolic relations provided by `Claripy`.

Gadget arrangement. The ROP chain compiler then determines *arrangements* of gadgets that can be used to perform *high-level* actions. For example, a gadget that pushes data to the stack can be paired with a gadget that pops data to create an arrangement that moves data from one register to another.

Payload generation. After the ROP compiler identifies the requisite set of gadget arrangements, it combines these gadgets into a *chain* to carry out high-level actions (such as executing attacker-specified system calls with specified arguments). This is done by writing gadget arrangements into a program state in `angr`, constraining their outputs to the provided arguments, and querying the SMT solver for a solution for their inputs.

Our implementation differs from Q in minor ways. First, Q made no use of the stack as scratch storage space. It is not clear why this is: one explanation is that their analysis platform did not support the modeling of stack operation, while another is that the approach remains more general if we assume that the stack is *not* necessarily pointed to by the stack pointer (and, thus, in an unknown location). In our integrated system, we could identify whether the stack pointer was pointing to the stack, since we had this metadata from the exploit that we generated with our implementation of AEG.

Another improvement has to do with the gadget classification. Q used a *value sampling* method to identify specific classes of gadgets, which led to some number of missed gadgets chains due to the limited coverage of the sample

Technique	Based On	Described In
Dynamic Symbolic Execution	*Various* [12], [16], [20]	IV-A2, IX
Veritesting	Veritesting [5]	IV-A2, IX
Under-constrained DSE	UCSE [46]	IV-A2, X
Symbolic-Assisted Fuzzing	Driller [54]	IV-A1, XI
Static Analyses	VSA [6]	III-C, VIII
Crash Replay	Replayer [43]	V-A, XII
Exploit Generation	AEG [4]	V-B, XIII
Exploit Hardening	Q [48]	V-C, XIV

TABLE I
ANALYSES IMPLEMENTED AND EVALUATED IN THIS PAPER, THE LITERATURE ON WHICH THEY ARE BASED, AND THE SECTIONS OF THIS PAPER IN WHICH THEY ARE DISCUSSED.

CGC Qualifying Position	Binaries Crashed
First	77
Second	12
Third	57
Fourth	9
Fifth	23
Sixth	57
Seventh	44
Eighth (did not qualify)	39
Ninth (did not qualify)	65

TABLE III
NUMBER OF CRASHED BINARIES FOR THE TOP 9 COMPETITORS IN THE CGC QUALIFICATION EVENT.

data. In our approach, we symbolically analyze every gadget, using careful caching techniques to keep the analysis fast.

XV. COMPARATIVE EVALUATION

By leveraging `angr`'s design, we were able to reproduce the binary analysis techniques that we have discussed, on the same codebase, enabling a comparative evaluation of their effectiveness. To the best of our knowledge, this has not been done before: previous comparisons were carried out on different implementations, leaving the possibility of differences in results being introduced by implementation differences. With the exception of the fuzzer itself (AFL), our analyses are all implemented on the same analysis engine and share over 90% of the same code base with each other.

We use a corpus of CGC binaries, released by DARPA for the CGC Qualification Event, to carry out our evaluation. As discussed in Section II-B, these binaries vary widely in complexity, but utilize a simple environment model, designed by DARPA to reduce the implementation effort of analysis systems.

We evaluate the techniques that we implemented for CFG recovery, dynamic and static vulnerability discovery, crash replay, exploitation, and exploit hardening. A summary of the analyses we implemented and evaluated, along with the literature on which they are based and the sections in this paper in which they are described, is produced in Table I.

A. CFG recovery

As the CFG is used as a pre-requisite for other analyses in `angr`, it is important to understand how well `angr`'s CFG recovery performs. As we discussed in detail in Section VII, `angr` has two CFG recovery algorithms: `CFGAccurate` relies on a base approach of *forced execution* and provides two methods of indirect jump resolution (*backwards slicing* and *symbolic back-traversal*), while `CFGFast` mainly uses recursive disassembly and heuristics to quickly identify functions and inter-function control flow.

To understand the effectiveness of these recovery techniques, we compared `CFGFast` and `CFGAccurate` against the CFG recovery of a state-of-the-art commercial tool, IDA Pro 6.9, on CGC binaries. While little details about how IDA Pro recovers the CFG are available, based on descriptions in previous work [59] as well as our observations, we believe that IDA Pro disassembles a binary recursively, uses symbols and other heuristics to determine locations of functions throughout a binary, and then utilizes some lightweight data-flow analyses to further solve the targets of indirect jumps. This makes it more similar, conceptually, to `CFGFast` than to `CFGAccurate`. As ground truth CFG information is not available, we evaluate our results in terms of the relative number of recovered basic blocks and control flow transfers between the results of IDA's and our CFG recovery.

We first evaluate the completeness of our CFG, comparing the blocks and edges identified by `CFGFast` and the graph generated by IDA Pro. Table II shows our results. `CFGFast` has a slightly better code coverage than IDA Pro, and recovers more edges. We believe that this is because the lightweight data-flow analyses and heuristics that are used by `CFGFast` are more advanced than those used by IDA. Manual analysis of recovery results on a few binaries indicates that `CFGFast` is more aggressive in terms of code recovery: while IDA Pro believes certain parts of code are not reachable and refuses to disassemble it as code, `CFGFast` identifies such locations as code. A possible explanation for this is that our approach might be *overly* aggressive, and as such, it might mis-identify such locations. However, we have not identified such cases when analyzing CGC binaries.

As some binary analyses require *reachability information* from the entry point, we have also included a comparison against the *reachable* portion of a CFG generated by IDA Pro (that is, a CFG comprising those blocks for which a path from the entry point can be determined) against the CFG recovered by `angr`'s `CFGAccurate` analysis. Table II shows our results. By improving the forced execution technique with backward slicing, `angr` substantially improves its ability to reconstruct the CFG. However, since `CFGAccurate` does not leverage *ad hoc* heuristics, the resulting CFG's code coverage is not as high as IDA Pro's. To achieve a better coverage, the user can provide `CFGAccurate` with all recovered functions from `CFGFast` as starting points.

B. Evaluation of Vulnerability Analysis Techniques

In Sections VIII through XI, we describe the implementation of several vulnerability discovery techniques. Here, we present the result of a comparative evaluation of these techniques as applied to the CGC dataset. We ran these evaluations with a timeout of 24 hours, which is the time period of the DARPA competition from which we retrieved the evaluation dataset.

We provide a summary of these results in Table IV. Additionally, to provide a better context for the number of

Approach	Functions		Function Edges		Blocks		Block Edges		Bytes		Time (s)	
	M	A	M	A	M	A	M	A	M	A	M	A
IDA Pro 6.9	48	52.96	76.5	99.62	829	3589.93	1188	6487.68	14037	104779.66	1.14	1.80
`angr` - `CFGFast`	61	70.08	88	118.74	843	3609.45	1193	6538.52	14296	105007.49	0.87	5.01
IDA Pro 6.9 - reachability	37	40.96	74	90.76	496	1043.81	759	1693.01	7874	21721.85	1.14	1.80
`angr` - forced execution	31	33.24	48	55.22	349.5	413.85	612	751.96	6125	13963.5	23.50	36.96
`angr` - symbolic back traversal	32	33.76	50	56.28	368	635.41	645	1089.78	6323	10883.51	27.22	34.10
`angr` - backward slicing	30	32.80	47.5	53.89	344.5	653.56	594	1178.98	6109.5	14641.85	24.78	79.46

TABLE II
EVALUATION OF `CFGFAST`'S AND `CFGACCURATE`'S RECOVERED CFG VERSUS THE CFG RECOVERED BY IDA PRO. THE MEDIAN NUMBER (M) AND AVERAGE NUMBER (A) OF EACH VALUE ACROSS ALL BINARIES ARE SHOWN.

Technique	Replayable	Semantic Insight	Scalability	Crashes	False Positives
Dynamic Symbolic Execution	Yes	High	Low	16	0
Veritesting	Yes	High	Medium	11	0
Dynamic Symbolic Execution + Veritesting	Yes	High	Medium	23	0
Fuzzing (AFL)	Yes	Low	High	68	0
Symbolic-Assisted Fuzzing	Yes	High	High	77	0
VSA	No	Medium	High	27	130
Under-constrained Symbolic Execution	No	High	High	25	346

TABLE IV
EVALUATION RESULTS ACROSS ALL VULNERABILITY DISCOVERY TECHNIQUES.

crashes identified by our techniques, we have included the number of crashes identified by the competitors at the actual CGC Qualification Event, in Table III. The overall scores of the teams relied on more than just crash counts, so the placement in the qualifying event is not correlated with the position of the competitors. Two of these competitors, the first-place team [27] and the seventh-place team [57], have written blog posts describing their techniques in the competition. Both teams used a symbolically-assisted fuzzing technique, conceptually similar to Driller. Note that, while our implementation of Driller identifies the same number of vulnerabilities as the first place team, this is a coincidence (likely driven by the similarity between the techniques).

Dynamic symbolic execution. We chose to evaluate dynamic symbolic execution both alone and in the presence of the Veritesting path explosion mitigation technique. We describe the implementation details of these approaches in Section IX.

As expected, dynamic symbolic execution frequently succumbed to the path explosion problem. In total, the standard approach identified vulnerabilities in 16 of the CGC binaries. Veritesting, which is designed to partially mitigate the path explosion problem, identified only 11, for a combined count of 23 applications in which vulnerabilities were identified.

We were initially surprised to find that, despite the better results, the Veritesting approach found less vulnerabilities than dynamic symbolic execution alone. Investigating these four binaries, we identified an interesting trade-off inherent to Veritesting. Veritesting uses efficient path merging to combat path explosion, which is responsible for its ability to explore deeper paths in the binary before path explosion renders further progress impossible. However, such path merging introduces complex expressions (e.g., if the value of register `eax` differs between two merged paths, the value of the merged path must be a complex expression encoding both previous values) and overloads the constraint solver. Thus, the solve times of the constraint solver tend to increase as more and more of these merges are done. As constraint solving is an NP-complete problem, the increased complexity leads to vulnerabilities becoming unreachable within a reasonable time. The result of this is that Veritesting is able to identify shallow bugs that dynamic symbolic execution otherwise experiences a path explosion with, but overwhelms the constraint solver for longer paths.

Symbolic-assisted fuzzing. Assisted fuzzing has proven to be extremely effective in the literature. In Section XI, we discuss an implementation of a symbolic-assisted fuzzing method, dubbed Driller [54].

This symbolic-assisted fuzzer uses AFL for the fuzzing component. Each input that AFL produces is traced in the dynamic symbolic execution engine to identify code sections that could be reached by careful mutation of the input. This careful mutation is carried out by the symbolic constraint solver, and the input is reintroduced to AFL for further execution and mutation. Because the individual inputs traced by the DSE engine do not branch (as all the input is concrete), there is no path explosion during tracing, and AFL limits the number of inputs passed to the DSE engine by filtering out all the inputs that do not increase code coverage.

It should be mentioned that AFL alone is able to identify vulnerabilities in a significant amount of the CGC services. In fact, of the 77 vulnerabilities that our symbolic-assisted fuzzer detected, 68 were detected by AFL alone. The remaining 9 were found through the use of symbolic assistance.

DSE vs. fuzzing. The difference between the results of the various dynamic symbolic execution approaches are surprising. One might reasonably expect DSE to identify roughly as many vulnerabilities as symbolically-assisted

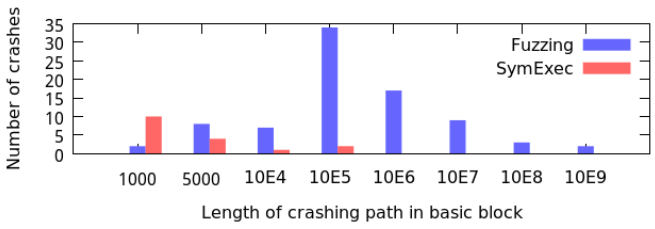

Fig. 1. Length of crashing paths discovered by fuzzing vs. dynamic symbolic execution.

fuzzing, and more than fuzzing alone. In reality, fuzzing identified almost *three times* as many vulnerabilities. In a sense, this mirrors the recent trends in the security industry: symbolic analysis engines are criticized as impractical while fuzzers receive an increasing amount of attention. However, this situation seems at odds with the research directions of recent years, which seem to favor symbolic execution.

Analyzing the crashing inputs that these approaches did find, we identified an interesting result: the exploits found by dynamic symbolic execution engines tend to represent *short* paths. This result is presented in Figure 1. By spot-checking several applications where dynamic symbolic execution (even with Veritesting) *failed* to find vulnerabilities, we have concluded that this is due to an increase in analysis complexity, exponentially proportional to the length of the path.

Specifically, given a path A, there is a chance p_a that A will split at the end of the next conditional jump, and A_1 will follow the path that takes the jump, while A_2 will follow the path that does not. At the next conditional jump, there is a chance that A_1 and A_2 will fork as well. Thus, the amount of resulting paths to analyze increases exponentially, and the chance that an unreasonable number of paths will have to be analyzed at any point is exponentially proportional to how many basic blocks have been executed by the analysis. As a result, the typical dynamic symbolic execution approach is best for finding *shallow* crashes that do not require the execution of many basic blocks. *Deep* crashes, on the other hand, tend to be hidden and made unreachable by the path explosion.

To further understand the relative effectiveness of the techniques, we calculated the *code coverage* of the generated test cases. We found that symbolic execution (including Veritesting) covered an average of 330 blocks per binary (with a median of 260), while fuzzing covered 689 (with a median of 402) and symbolic-assisted fuzzing covered 698 (with a median of 406). These results yield another interesting conclusion: if the paths generated by fuzzing or symbolic-assisted fuzzing were combined into a *graph*, it would represent a CFG with more code coverage than the one recovered by `CFGAccurate` (and, by virtue of each edge in the graph being reachable by definition, with perfect completeness), implying a need for further improvement of the accurate CFG recovery algorithm.

Under-constrained symbolic execution. We extended `angr` to support under-constrained symbolic execution to better understand how effective such techniques are on our dataset. These details are presented in Section X.

UC-`angr` reported 371 vulnerabilities in the CGC binaries. However, as this approach analyzes functions without their context, it suffers from similar problems as static analyses: the results contain a large number of false positives, and the results are not replayable (that is, they do not generate crashing inputs, but instead point out the location of vulnerabilities). In fact, we identified 346 false positives in UC-`angr`'s results, leaving 25 true positives and resulting in a false positive rate of 93%, which is in line with those reported by UC-KLEE [46].

Static buffer overlap detection. To be able to compare the different types of vulnerabilities identified by fuzzing, symbolic execution, and other static analyses, we implemented a VSA-based memory corruption detection analysis. We describe it in detail in Section VIII.

Similarly to UC-`angr`, our VSA results are not replayable and suffer from false positives. In total, VSA was able to identify 27 actual vulnerabilities in CGC binaries while producing 130 false positives, resulting in a false-positive rate of 82.8%.

Non-replayable vs replayable analyses. Another surprising result is the comparatively low performance of non-replayable techniques (VSA and under-constrained symbolic execution). While these techniques, freed from the replayability requirement, can achieve more coverage in their analysis, we found that the context they lacked resulted in an enormous amount of false positives in this dataset. To keep the false positive rate reasonable, we had to implement aggressive false positive filtering (as discussed in Section X), which filtered out many true positives as well.

The improvement of static analysis techniques on real binaries appears to be an area in need of research attention, and we are considering it as a direction for future work.

C. Exploitation Evaluation.

After a crash is identified by the above approaches, we attempt to replay and exploit it to understand its severity.

Crash replay. As we discuss in Section V-A, crashing inputs identified by vulnerability discovery analyses might not be trivially replayable due to environmental data (such as the random seed) having been *de-randomized* by the analysis. We analyzed crashes for each CGC binary, using the reference crashing inputs provided by DARPA for binaries where we were unable to identify vulnerabilities with our vulnerability identification techniques. Of these crashing inputs, 6 were not trivially replayable. That is, rather than simply replaying the crashing input provided to us by the vulnerability identification engines, we had to re-analyze the interaction with the binary to recover challenge-response components present in these binaries.

Interestingly, DARPA imposes a limitation on the authors of CGC binaries from the CGC Qualifying Event that disallows control flow from being impacted by random data. This means that the limitation of Replayer discussed in Section XII, the introduction of different path predicates due to different values of random data, does not apply to its operation on CGC binaries. Though `angr` did hang on one of the applications, manual analysis revealed this to be an implementation issue,

rather than one with the approach and, as expected, Replayer was able to recover the input specification of the remaining 5.

While 6 binaries are not a large dataset, this result suggests that current techniques in this area are able to adequately handle binaries in the absence of control flow variance caused by random data. Further work is needed to evaluate, and possibly extend, these techniques on real-world binaries with more complex control flow.

Automatic exploit generation. After identifying the crash and running it through Replayer, we are left with an input specification that reliably crashes the target application. However, such inputs might still not be *exploitable*. For example, crashes caused by null pointer dereferences, of which there are many in the CGC dataset, are not exploitable on modern systems. To separate exploitable from non-exploitable inputs, we attempt to generate an exploit from the crash.

We attempted to automatically generate exploits for all of the CGC applications, using techniques proposed in the AEG system [4]. However, we were surprised to find that only *4* crashing exploits could be weaponized into exploits using these techniques. Looking deeper into the binaries, we understood why. First, the goal of the CGC Qualification Event was to find *crashes* for the binaries, not exploits. As such, many of the vulnerabilities in these binaries are not actually exploitable (i.e., null-pointer dereferences). Second, as the CGC binaries model a wide range of realistic exploitation scenarios, we found that the techniques proposed by AEG were not applicable to the majority of them.

The current state of the art in this field is fairly basic, and it appears in these results. Further research is required into this field to enable the automatic exploitation of complex vulnerabilities.

Exploit hardening. Even an exploitable vulnerability might be mitigated by modern protections. As a result, *exploit hardening* is required, and has been investigated by recent work. We reimplemented the techniques proposed by Q [48] and attempted to harden the exploits generated by AEG.

The Q implementation was able to harden 2 of the 4 exploits that AEG generated. Our analysis as to why the remaining two exploits could not be hardened revealed that the Q approach does not utilize enough information in the binary. In these two examples, there is not enough attacker-controlled data on the stack and a *stack pivot* is required to use attacker-controlled data in other parts of the program. The Q approach has no basis for reasoning about such operations and, as a result, these exploits cannot be hardened.

XVI. Conclusions

In this paper, we presented `angr`, a system that implements, in a unified framework, a number of techniques for the automated identification and exploitation of vulnerabilities in binaries. We presented, in a systematized fashion, the different analyses and the challenges we encountered when including them in our framework. By implementing these approaches in a single system, we were able to meaningfully compare their effectiveness on a dataset that was created for the evaluation of these techniques. The results of this evaluation can be used as a basis to highlight research directions, and to improve existing techniques.

We made `angr` open-source, so that the community can build on top of it and focus on addressing open challenges in the field of binary analysis.

Acknowledgements. This work is sponsored by DARPA under agreement number N66001-13-2-4039 and by the ONR under agreement number N00014-15-1-2948. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[2] OWASP Top 10 Project. http://http://www.owasp.org.

[3] The XcodeGhost malware. http://www.apple.com/cn/xcodeghost/#english.

[4] T. Avgerinos, S. K. Cha, B. L. Tze Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[5] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. pages 1083–1094, 2014.

[6] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4171 LNCS:202–213, 2008.

[7] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium*, pages 845–860, 2014.

[8] C. Barrett, L. De Moura, and A. Stump. SMT-COMP: Satisfiability modulo theories competition. In *Computer Aided Verification*, pages 20–23. Springer, 2005.

[9] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. A Taint Based Approach for Smart Fuzzing . In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 818–825. IEEE, 2012.

[10] Bloomberg Business. Hospital Gear Could Save Your Life or Hack Your Identity. http://www.bloomberg.com/features/2015-hospital-hack/.

[11] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 LNCS, pages 351–366. Springer Berlin Heidelberg, 2008.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, volume 8, pages 209–224, 2008.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[14] G. Campana. Fuzzgrind: un outil de fuzzing automatique. In *Actes du 7ème symposium sur la sécurité des technologies de linformation et des communications (SSTIC)*, pages 213–229, 2009.

[15] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song. Transformation-aware Exploit Generation using a HI-CFG. Technical report, DTIC Document, 2013.

[16] 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*, pages 380–394, 2012.

[17] S. K. Cha, M. Woo, and D. Brumley. Program-Adaptive Mutational Fuzzing. In *Proceedings of IEEE Symposium on Security and Privacy*, volume 2015-July, pages 725–741, 2015.

[18] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 20–29. ACM, 2011.

[19] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective Symbolic Execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability*, 2009.

[20] 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*, pages 265–278, 2011.

[21] C. Cifuentes and M. Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *Proceedings of the Seventh International Workshop on Program Comprehension*, pages 192–199. IEEE, 1999.

[22] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol Specification Extraction. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, pages 110–125. IEEE, 2009.

[23] DAPRA. DARPA Cyber Grand Challenge. http://www.cybergrandchallenge.com/.

[24] DARPA. CyberGrandChallenge samples git repository. https://github.com/CyberGrandChallenge/samples.

[25] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] D. Engler and D. Dunbar. Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 1–4, 2007.

[27] ForAllSecure. Unleashing the Mayhem CRS. http://blog.forallsecure.com/2016/02/09/unleashing-mayhem/.

[28] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 40, pages 213–223, 2005.

[29] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *ACM Queue*, 10(1):20, 2012.

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

[31] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong. CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 78–87. IEEE, 2012.

[32] Indefinite Studies. The Halting Problem for Reverse Engineers. http://indefinitestudies.org/2010/12/19/the-halting-problem-for-reverse-engineers/.

[33] J. Kinder and H. Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th international conference on Computer Aided Verification*, pages 423–427, Berlin, 2008. Springer-Verlag.

[34] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium*, volume 13, pages 18–18, 2004.

[35] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, page 193, 2012.

[36] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[37] Y. Li, Z. Su, L. Wang, and X. Li. Steering Symbolic Execution to Less Traveled Paths. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object Oriented Programming Systems Languages & Applications*, pages 19–32, 2013.

[38] 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.

[39] C. Miller. Fuzzing With Code Coverage By Example. https://fuzzinginfo.files.wordpress.com/2012/05/cmiller_toorcon2007.pdf.

[40] M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis Through Linear Algebra. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 39, pages 330–341, 2004.

[41] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7705 LNCS, pages 115–130, 2012.

[42] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos. The BORG: Nanoprobing Binaries for Buffer Overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 87–97. ACM, 2015.

[43] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic Protocol Replay by Binary Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 311–321, 2006.

[44] F. Pérez and B. E. Granger. IPython: A System for Interactive Scientific Computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. http://ipython.org.

[45] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, volume 2015-July, pages 709–724, 2015.

[46] D. a. Ramos and D. Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Security Symposium*, pages 49–64, 2015.

[47] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-Extended Symbolic Execution on Binary Programs. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, page 225, 2009.

[48] E. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, volume 8, page 25, 2011.

[49] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[50] B. Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. In *Proceedings of Ninth working conference on Reverse engineering, 2002*, pages 45–54. IEEE, 2002.

[51] D. K. Sean Heelan. *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*. PhD thesis, University of Oxford computing laboratory, 9 2009.

[52] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, volume 22, pages 552–561, 2007.

[53] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of Network and Distributed System Security Symposium*, number February, pages 8–11. Internet Society, 2015.

[54] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[55] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[56] K. Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, Aug. 1984.

[57] Trail of Bits Blog. How We Fared in the Cyber Grand Challenge. http://blog.trailofbits.com/2015/07/15/how-we-fared-in-the-cyber-grand-challenge/.

[58] J. Troger and C. Cifuentes. Analysis of Virtual Method Invocation for Binary Translation. In *Proceedings of Ninth Working Conference on Reverse Engineering, 2002*, pages 65–74. IEEE, 2002.

[59] L. Xu, F. Sun, and Z. Su. Constructing Precise Control Flow Graphs from Binaries. *University of California, Davis, Tech. Rep*, 2009.

[60] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[61] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic Inference of Search Patterns for Taint-style Vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, volume 2015-July, pages 797–812, 2015.

[62] M. Zalwski. Bunny the Fuzzer Documentation. http://code.google.com/p/bunny-the-fuzzer/wiki/BunnyDoc.