2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)

Mockingbird: A Framework for Enabling Targeted Dynamic Analysis of Java Programs*

Derrick Lockwood Benjamin Holland Suresh Kothari
Department of Electrical and Computer Engineering, Iowa State University
Ames, IA 50011 {djlock,bholland,kothari}@iastate.edu

***Abstract*—The paper presents the Mockingbird framework that combines static and dynamic analyses to yield an efficient and scalable approach to analyze large Java software. The framework is an innovative integration of existing static and dynamic analysis tools and a newly developed component called the Object Mocker that enables the integration. The static analyzers are used to extract potentially vulnerable parts from large software. Targeted dynamic analysis is used to analyze just the potentially vulnerable parts to check whether the vulnerability can actually be exploited.**

We present a case study to illustrate the use of the framework to analyze complex software vulnerabilities. The case study is based on a challenge application from the DARPA Space/Time Analysis for Cybersecurity (STAC) program. Interestingly, the challenge program had been hardened and was thought not to be vulnerable. Yet, using the framework we could discover an unintentional vulnerability that can be exploited for a denial of service attack. The accompanying demo video depicts the case study.

Video: https://youtu.be/m9OUWtocWPE

***Index Terms*—Static analysis, Dynamic Analysis, Fuzzing, Software Vulnerability**

I. Introduction

We present the Mockingbird framework; it combines static and dynamic analyses using a *statically-informed dynamic analysis* [1] approach. This approach suits analysis tasks that can be subdivided into: (a) extracting relevant code buried in large and complex software, and (b) verifying specified properties with respect to the relevant code. An important application is detecting *side channel* (SC) and *algorithmic complexity* (AC) vulnerabilities [2]. The paper [3] describes how statically-informed dynamic analysis can be applied for detecting such open-ended vulnerabilities. We demonstrate the use of the Mockingbird framework for detecting AC vulnerabilities (e.g., Billion Laughs [4]). Combining static and dynamic analyses opens the possibility to have the best of both worlds - cover all execution paths and yield precise results. However, how to do it effectively is a challenge [5]–[7].

The Mockingbird framework is an innovative integration of existing static and dynamic analysis tools and a newly developed component *Object Mocker* that enables the integration. Specifically, the Mockingbird framework incorporates:

- Existing static analyzers built using the Atlas platform [1], [8]–[10]. These analyzers are used to extract the relevant code - the code that is potentially vulnerable.
- The existing AFL fuzzer [11] supplemented by the Kelinci, a tool that interfaces AFL to operate on Java programs [12].
- The Object Mocker that can automatically create harnesses to apply the AFL fuzzer to dynamically analyze the relevant code extracted using the static analyzers.

The key novelty of the Mockingbird is *targeted dynamic analysis* (TDA). TDA performs dynamic analysis of just the relevant code using binary inputs. The framework provides the capability to mock any Java object type and produces a testing harness that transforms binary inputs to appropriate object types. For fuzzers such as AFL, users manually create a test harness that translates the binary inputs from the fuzzer to the data structures expected by the target program. Developing the harnesses manually for just the relevant code is difficult and laborious because the program state must be considered as the input. The program state is communicated to the relevant function via the *stack memory* using function parameters and return values or through the *heap memory* using reads and writes to global variables. The harness incorporates mocked objects that mimic the program artifacts that carry the inputs into the relevant code. The Mockingbird framework creates the harnesses automatically. The framework earns its name from the term "mock objects" as used by the testing community [13].

The Mockingbird framework optionally allows constraints to be placed on the values of object fields that store the encapsulated program data. With this enhancement, it is possible to systematically incorporate domain-specific knowledge into the automatically generated test harnesses. For example, if a program only operates on byte arrays that begin with a magic sequence, such as `0xFFD8` in the case of JPEG file formats, then the test harness can simply prefix the program input with the known sequence to increase the chances of quickly driving the program execution in meaningful ways. The Mockingbird framework supports configurable input generation constraints to be specified for test harnesses as a means for injecting domain knowledge.

We present a case study in which the Mockingbird frame-

* This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0080 and FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

2574-1934/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-Companion.2019.00033

works is used to detect the AC vulnerability in a challenge application from the DARPA's STAC program.

II. A Motivating Example for Mocking

Mock objects are simulated objects that mimic the behavior of real objects in controlled ways. Mock objects serve two important purposes: 1) they allow direct control of the data values that could have reached the relevant code, and 2) they effectively isolate the relevant code by breaking existing control and data dependencies (replacing an object with a mock object removes dependencies since mock objects have no program dependencies).

The need for mocking is illustrated with a simple code example shown in Listing 1. We shall also use this example to illustrate how the test harness can be generated automatically.

Listing 1. Motivating Example

```
1  public class Example {
2    public static boolean isVowel(char c) {
3      return c == 'a' || c == 'e' || c == 'i'
4          || c == 'o' || c == 'u' || c == 'y';
5    }
6    class Pet {
7      private String name;
8      public Pet(String name) {
9        this.name = name;
10       sleep(5000);
11     }
12     public String getName() {
13       return name;
14     }
15     public double getVowelRatio() {
16       double vowels = 0;
17       String name = getName().toLowerCase();
18       for(char c : name.toCharArray()) {
19         if(isVowel(c)) {
20           vowels++;
21         }
22       }
23       return vowels / (name.length() - vowels);
24     }
25   }
26 }
```

The `getVowelRatio` method has a division-by-zero vulnerability that occurs when the input consists of only vowels. In order to perform a TDA of the `getVowelRatio` method using existing tools such as Kelinci [12], which adapts the AFL fuzzer [11] to execute Java programs, a test harness would have to be developed that constructs instances of the `Pet` class with desired string values to test. Furthermore, constructing the objects could involve code that is unnecessary for mocking but it consumes significant execution time. The "sleep" on line 10 symbolizes such unnecessary code. By mocking the instance variable "`name`", we can directly control the value of "`name`" and elide the expensive constructor logic.

III. Mockingbird Framework

Figure 1 depicts an overview of the Mockingbird framework architecture and workflow.

As shown in Figure 1, a variety of program analyzers are used to extract the relevant code given a class of vulnerabilities. The static program analyzers are designed to be

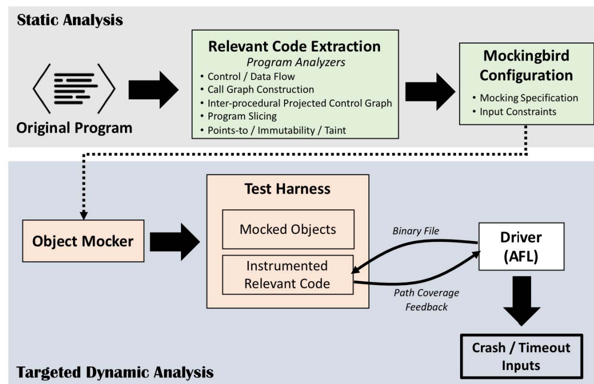

Fig. 1. Mockingbird Framework Architecture and Workflow

configurable to cater to different classes of software vulnerabilities and the analysis results in the inter-procedural *projected control graph* (PCG) that succinctly captures relevant program behaviors [10], [14], [15].

These program analyzers are built on top of Atlas [8], a graph database platform with graphs as the unifying abstraction to build program analyzers for a variety of languages including Java and Java bytecode. The case study in Section IV illustrates how these program analyzers are used to extract the relevant code from large and complex software.

As shown in Figure 1, the relevant code feeds into the Mockingbird's configuration component. This component builds a configuration file, which serves as the specification for creating the harness. For the code shown in Listing 1, the `getVowelRatio` instance method in the `Pet` class is extracted as the relevant code. The corresponding configuration file is shown in Listing 2.

Listing 2. An Example of Mockingbird Configuration

```
1  {
2    "definition": {
3      "class": "Example$Pet",
4      "method": "getVowelRatio",
5      "instance_variables": [
6          {
7              "name": "name"
8          }
9      ]
10   },
11   "config": {
12     "timeout": 1000
13   }
14 }
```

The configuration file is on purpose, human-readable so that a human operator can easily refine the mocking specifications. As mentioned earlier, one purpose is to enable injection of domain knowledge to constrain the inputs to improve the TDA efficiency. It also provides an opportunity to manually verify and alter the mocking.

As shown in Figure 1, the Object Mocker creates the test harness that incorporates the mocked objects - as specified by the configuration file. Internally, the Object Mocker leverages the bytecode manipulation capabilities provided by

ByteBuddy [16] and the ubiquitous ASM [17] bytecode manipulation library to generate runtime compatible mock objects containing only the fields and methods required to execute the relevant code. With the configuration file shown in Listing 2, the `Pet` object is mocked, without calling the constructor, by providing a simulated instance variable called "`name`" (the fact that "`name`" is a `java.lang.String` type is picked up automatically by the framework using Java reflection).

Pure methods (methods that do not mutate pre-existing program state) as identified by [9] are not mocked. For the motivating example, the `isVowel` method is correctly identified as pure and automatically excluded from mocking. By default, the JDK is not mocked.

As shown in Figure 1, after the harness is created the rest of TDA can be performed with AFL. The AFL iteratively generates new binary inputs through mutations guided by a genetic algorithm to mutate the binary inputs to maximize path coverage through a feedback loop.

Importantly, the Mockingbird framework can use another fuzzer in place of AFL. Moreover, a fuzzer such as AFL can be further enhanced by creating custom tools that can work with the fuzzer. The Mockingbird capability to programmatically control the data of each mocked field, is valuable for developing custom dynamic analysis tools. This is brought out in the case study in Section IV. Combined with Daikon [18], the Mockingbird framework can be used to compute the invariants with respect to the relevant code and not the entire program [19]. In our study, the program invariants are computed for the potentially vulnerable code. As we will show in the case study in Section IV, the invariants can be valuable to gain a holistic understanding of the root cause of the vulnerability.

A tool called afl-unicorn [20] creates test harnesses and performs targeted dynamic analysis but it works only for assembly languages and does not address the problem of mocking objects. Several object mocking tools exist, such as TestNG and Mockito [21], but we found that these tools made many low level assumptions that prevented the direct control of object states that we required for generalized dynamic analysis.

Our implementation is open source and freely available online at: https://github.com/kcsl/Mockingbird.

IV. Case Study

We examine a DARPA STAC challenge application called BraidIt for AC vulnerabilities. The application's source code is available at: https://github.com/Apogee-Research/STAC.

The case study demonstrates: (1) the use of Mockingbird framework to efficiently discover an algorithmic complexity vulnerability, and (2) a follow-up experiment using Mockingbird's APIs to create a custom tool for computing program invariants that provide holistic understanding of the vulnerability. The follow-up experiment reveals not just one input but a class of inputs that cause the vulnerability; in particular, it reveals the smallest input that can cause the vulnerability.

DARPA's BraidIt application is described as a two player peer-to-peer game where players' attempt to recognize topologically equivalent braids (also known in mathematics as an Artin braid group [22]). For the demonstration purpose, we have chosen a relatively small application with 5,844 lines of decompiled Java code (not including any of the 17 third-party library dependencies). Static analysis reveals that the application contains 51 loops, of which 29.4% of the loops are contained in a single class called `Plait`.

To find the relevant code for an AC time vulnerability, the loop analyzer [23] is used to locate loops with complex termination conditions. The set of loops is further refined by a reachability analyzer that retains only those loops whose termination conditions can be affected by external inputs. This leads to a loop in the `normalizeCompletely` method of the `Plait` class. The loop is shown in Listing 3.

Listing 3. Relevant Code of Vulnerable Loop

```
1  public void normalizeCompletely() {
2      this.freeNormalize();
3      while (!this.isReduced()) {
4          this.normalizeOnce();
5          this.freeNormalize();
6      }
7  }
```

The loop's termination condition depends on the result of the `isReduced` method. A data flow analysis of the `isReduced` method reveals that: 1) the termination condition consists of a complex series of string manipulations of a global variable called `intersections`, which `freeNormalize` and `normalizeOnce` both mutate in every loop iteration, and 2) the `intersections` variable is reachable from the outside.

Moreover, a static analyzer for locating code relevant for AC space vulnerability also reports the loop shown in Listing 3 because the loop children `normalizeOnce` and `freeNormalize` write to the file system during each iteration of the loop. With incriminating evidence gathered from various static analyzers, the code shown in Listing 3 becomes a prime target for TDA.

With the help of static analyzers, we have found the relevant code - it is small but extremely complex code. It is quite difficult to reason about this code manually or by using static analysis. In fact, between the two methods operating on the `intersections` variable there are 9 unique string operations in 62 locations, 20 of which are within loops as well as 8 unique character level operations in 60 locations, of which 27 locations are within loops. This is where TDA can be immensely helpful. The `isReduced` method computes a non-trivial property on the `intersections` string, which is changing during each loop iteration. The question is: can there be a string that has the property of being irreducible, and therefore cause an infinite loop that writes to the file system? If true, it is actually an algorithmic complexity vulnerability in time as well as space and thus an opportunity for the attacker to launch an extremely effective denial of service attack.

Running the experiment using the Mockingbird framework yielded in 20 hours the first input that actually triggers the AC time vulnerability; additional 22 inputs were found in 39 hours. The first input was "ËĎġčçęêďªã;" it hints that the vulnerability has to do with Unicode localization, but it does not explain the root cause of the vulnerability.

A. A Follow-up Experiment

The motivating questions for this follow-up experiment are: *Why is the application vulnerable to particular string inputs? Is there actually a class of inputs for the exploit?* Answers to such questions are important to gain holistic understanding of the vulnerability and modify the program to eliminate the root cause of the vulnerability.

Using Mockingbird's APIs, we can quickly build a custom tool to help answer these questions. In this experiment, we targeted the same relevant code and retained the same mocking specifications, but replaced the AFL driver on the front-end with a brute-force search of the alphabet of characters accepted by the `Plait` constructor. Within 20 minutes the brute-force search revealed the string value "ªªª" as the minimal string for the exploit. The Daikon invariant detector reveals that all previously discovered strings share a common substring ('ª').

Debugging the application with this minimal input reveals the heart of the vulnerability. It reveals that the `freeNormalize` method removes a pair of matching characters (leaving a single 'ª' character remaining). Next the `isReduced` method returns false if the string contains an uppercase character of a lowercase character. This scheme is perfectly reasonable for ASCII characters, but there exists a small set of alphabetic Unicode characters such as 'ª' where uppercase('ª')==lowercase('ª'). Since a string of a single lowercase character that is identical to its uppercase is never reducible and the `freeNormalize` method will never affect a string of one character, therefore the loop will never terminate.

After getting the holistic understanding, we searched for previous studies of such vulnerabilities and found that these types of vulnerabilities are known to abound in codes for localization (which is exactly what the relevant code in the case study is for) and actually a patent is granted to the Honeywell Corporation for a method for detecting, analyzing, and mitigating similar vulnerabilities [24].

V. Conclusion

Broadly speaking program analysis techniques can be divided into two main camps: static analysis and dynamic analysis. Static analysis can be efficient but it is imprecise. Dynamic analysis is precise but it is inefficient. Because of diametrically opposite tradeoffs, combining static and dynamic analyses opens the possibility to have the best of both worlds. However, how to do combine static and dynamic analyses effectively is a challenge. The Mockingbird framework is designed to address this challenge. The framework represents a major engineering effort involving integration of existing static analysis and fuzzing tools and the development of an innovative component Object Mocker. The Object Mocker can mock any Java object. A case study using a DARPA challenge application is presented to illustrate how the framework can be effectively applied in practice.

References

[1] B. Holland, G. R. Santhanam, P. Awadhutkar, and S. Kothari, "Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities," in *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference*. IEEE, 2016, pp. 79–84.

[2] DARPA, "Space/Time Analysis for Cybersecurity," https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-14-60/listing.html, 2014.

[3] G. R. Santhanam, B. Holland, S. Kothari, and N. Ranade, "Human-on-the-loop automation for detecting software side-channel vulnerabilities," in *International Conference on Information Systems Security*. Springer, 2017, pp. 209–230.

[4] M. Abliz, "Internet denial of service attacks and defense mechanisms," *University of Pittsburgh, Department of Computer Science, Technical Report*, pp. 1–50, 2011.

[5] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*. New Mexico State University Portland, OR, 2003, pp. 24–27.

[6] W. Le, "Segmented symbolic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 212–221.

[7] 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 *NDSS*, vol. 16, 2016, pp. 1–16.

[8] T. Deering, S. Kothari, J. Sauceda, and J. Mathews, "Atlas: A new way to explore software, build analysis tools," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014.

[9] B. Holland, G. R. Santhanam, and S. Kothari, "Transferring state-of-the-art immutability analyses: Experimentation toolbox and accuracy benchmark," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 484–491.

[10] B. Holland, P. Awadhutkar, S. Kothari, A. Tamrawi, and J. Mathews, "Comb: Computing relevant program behaviors." in *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*, ser. ICSE '18. IEEE Press, 2018.

[11] M. Zalewski, "american fuzzy lop (2.52b)," http://lcamtuf.coredump.cx/afl, Mar. 2018.

[12] R. Kersten, K. Luckow, and C. S. Păsăreanu, "Poster: Afl-based fuzzing for java with kelinci," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2511–2513.

[13] S. Freeman and N. Pryce, *Growing object-oriented software, guided by tests*. Pearson Education, 2009.

[14] A. Tamrawi and S. Kothari, "Projected control graph for accurate and efficient analysis of safety and security vulnerabilities," in *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*. IEEE, 2016, pp. 113–120.

[15] S. Kothari, P. Awadhutkar, A. Tamrawi, and J. Mathews, "Modeling lessons from verifying large software systems for safety and security," in *Proceedings of the 2017 Winter Simulation Conference*, 2017.

[16] "Byte buddy," http://bytebuddy.net, Sept. 2018.

[17] E. Kuleshov, "Using the asm framework to implement common java bytecode transformation patterns," *Aspect-Oriented Software Development*, 2007.

[18] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[19] B. Holland, "Computing homomorphic program invariants," Ph.D. dissertation, Iowa State University, 2018.

[20] N. Voss, "afl-unicorn: Fuzzing arbitrary binary code," https://github.com/Battelle/afl-unicorn, Jul. 2018.

[21] T. Kaczanowski, *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski, 2012.

[22] "Artin group," https://en.wikipedia.org/wiki/Artin_group, Spet. 2018.

[23] P. Awadhutkar, G. R. Santhanam, B. Holland, and S. Kothari, "Intelligence amplifying loop characterizations for detecting algorithmic complexity vulnerabilities," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 00, Dec. 2017, pp. 249–258.

[24] D. B. Kirk Schloegel, "Method for software vulnerability flow analysis, generation of vulnerability-covering code, and multi-generation of functionally-equivalent code," U.S. Patent US8 407 800B2, Mar. 26, 2013.