Static Program Analysis as a Fuzzing Aid

Bhargava Shastry[1](✉), Markus Leutner[1], Tobias Fiebig[1], Kashyap Thimmaraju[1], Fabian Yamaguchi[2], Konrad Rieck[2], Stefan Schmid[3], Jean-Pierre Seifert[1], and Anja Feldmann[1]

[1] TU Berlin, Berlin, Germany
bshastry@sec.t-labs.tu-berlin.de
[2] TU Braunschweig, Braunschweig, Germany
[3] Aalborg University, Aalborg, Denmark

Abstract. Fuzz testing is an effective and scalable technique to perform software security assessments. Yet, contemporary fuzzers fall short of thoroughly testing applications with a high degree of control-flow diversity, such as firewalls and network packet analyzers. In this paper, we demonstrate how static program analysis can guide fuzzing by augmenting existing program models maintained by the fuzzer. Based on the insight that code patterns reflect the data format of inputs processed by a program, we automatically construct an *input dictionary* by statically analyzing program control and data flow. Our analysis is performed before fuzzing commences, and the input dictionary is supplied to an off-the-shelf fuzzer to influence input generation. Evaluations show that our technique not only increases test coverage by 10–15% over baseline fuzzers such as *afl* but also reduces the time required to expose vulnerabilities by up to an order of magnitude. As a case study, we have evaluated our approach on two classes of network applications: nDPI, a deep packet inspection library, and tcpdump, a network packet analyzer. Using our approach, we have uncovered 15 zero-day vulnerabilities in the evaluated software that were not found by stand-alone fuzzers. Our work not only provides a practical method to conduct security evaluations more effectively but also demonstrates that the synergy between program analysis and testing can be exploited for a better outcome.

Keywords: Program Analysis • Fuzzing • Protocol Parsers

1 Introduction

Software has grown in both complexity and dynamism over the years. For example, the Chromium browser receives over 100 commits every day. Evidently, the scale of present-day software development puts an enormous pressure on program testing. Evaluating the security of large applications that are under active development is a daunting task. Fuzz testing is one of the few techniques that not only scale up to large programs but are also effective at discovering program vulnerabilities.

Unfortunately, contemporary fuzzers are less effective at testing complex network applications that handle diverse yet highly structured input. Examples of such applications are protocol analyzers, deep packet inspection modules, and firewalls. These applications process input in multiple stages: The input is first tokenized, then parsed syntactically, and finally analyzed semantically. The application logic (e.g., intrusion detection, network monitoring etc.) usually resides in the final stage. There are two problems that these applications pose. First, the highly structured nature of program input begets a vast number of control flow paths in the portion of application code where packet parsing takes place. Coping with diverse program paths in the early stages of the packet processing pipeline, and exploring the depths of program code where the core application logic resides is taxing even for state-of-the-art fuzzers. Second, the diversity of program input not only amplifies the number of control flows but also demands tests *in breadth*. For example, the deep packet inspection library, nDPI, analyzes close to 200 different network protocols [27]. In the face of such diversity, generating inputs that efficiently test application logic is a hard problem.

Although prior work on grammar-based fuzzing [13, 16, 29] partly address the problem of fuzz testing parser applications, they cannot be applied to testing complex third-party network software for two reasons. First, existing grammar-based fuzzers rely on a user-supplied *data model* or *language grammar* specification that describes the input data format. A fundamental problem with a specification-based approach to fuzzing is that the formal grammar of program input might not be available to begin with. Indeed, few network protocols have a readily usable formal specification. Therefore, grammar-based fuzzing at present, is contingent upon a data model that is—most often—manually created by an expert. Although proposals such as Prospex [7] that automatically create grammar specifications from network traces are promising, they are designed with a single protocol in mind. Automatic specification generation for diverse grammars has not been attempted. A second problem with certain grammar-based approaches that use whitebox testing is that they require significant software alterations, and rely on implementation knowledge. For example, to conduct grammar-based whitebox testing, parsing functions must be manually identified in source code, and detokenization functions must be written. Although *manual fallbacks* may be inevitable in the face of implementation diversity, prior approaches demand significant software revisions, making them ill-suited for security evaluation of *third-party* software.

In this paper, we demonstrate how the stated challenges can be addressed by augmenting fuzzing with static program analysis. Being program centric, static analysis can examine control flow throughout an application's codebase, permitting it to analyze parsing code in its entirety. This design choice makes our approach well-suited for testing complex network applications. Our approach has two key steps. First, we automatically generate a dictionary of protocol message constructs and their conjunctions by analyzing application source code. Our key insight is that code patterns signal the use of program input, and therefore sufficient cues about program input may be gathered by analyzing the source

code. To this end, we develop a static analyzer that performs data and control-flow analysis to obtain a dictionary of input constructs. Second, the dictionary obtained from the first step is supplied to an off-the-shelf fuzzer. The fuzzer uses the message fragments (constructs and conjunctions) present in the supplied dictionary toward input generation. Although anecdotal evidence suggests that a carefully constructed dictionary can dramatically improve a fuzzer's effectiveness [35], program dictionaries at present are created by a domain-specific expert. To make our analysis and test framework easily deployable on real-world code, we have developed a plugin to the Clang/LLVM compiler that can (i) Be automatically invoked at code compilation time, and (ii) Produce input dictionaries that are readily usable with off-the-shelf fuzzers such as afl. Indeed, our work makes security evaluations accessible to non-domain-experts e.g., audit of third-party code in the government sector.

We have prototyped our approach in a tool that we call Orthrus, and evaluated it in both controlled and uncontrolled environments. We find that our analysis helps reduce the time to vulnerability exposure by an order of magnitude for the `libxml2` benchmark of the fuzzer test suite [15]. Furthermore, we use Orthrus to conduct security evaluations of nDPI (deep packet inspection library), and tcpdump (network packet analyzer). Input dictionaries generated via static code analysis increase test coverage in nDPI, and tcpdump by 15%, and 10% respectively. More significantly, input dictionaries have helped uncover 15 zero-day vulnerabilities in the packet processing code of 14 different protocols in the evaluated applications that were not found by stand-alone fuzzers such as afl, and the Peach fuzzer. These results lend credence to the efficacy of our approach in carrying out security evaluations of complex third-party network software. Our prototype, Orthrus, is available at `https://www.github.com/test-pipeline/Orthrus`.

Contributions:

- To address the challenges of fuzzing complex network software, we propose a static analysis framework to infer the data format of program inputs from source code.
- We propose a novel approach—the use of static program analysis—to augment fuzzing. To this end, we couple our analysis framework with an off-the-shelf fuzzer.
- Finally, we prototype our approach and extensively evaluate its impact. Our prototype achieves an improvement of up to 15% in test coverage over state-of-the-art fuzzers such as afl, expedites vulnerability discovery by an order of magnitude, and exposes 15 zero-day vulnerabilities in popular networking software[4]. These results validate our proposition that static analysis can serve as a useful fuzzing aid.

[4] Ethical Considerations: Vulnerabilities found during our case studies have been responsibly disclosed to the concerned vendors who have subsequently patched them.

2 Background

In this section, we provide a brief overview of static analysis, and fuzz testing that is relevant to our work.

Static Analysis Our application of static analysis is closer to the notion of static analysis as a program-centric *checker* [10]: Tools that encapsulate a notion of program behavior and check that the implementation conforms to this notion. Historically, static analysis tools aimed at finding programming errors encode a description of correct (error-free) program behavior and check if the analyzed software meets this description. In contrast, our analyses encode input-processing properties of a program in order to extract features of the input message format.

Static analysis helps in analyzing the breadth of a program without concrete test inputs. However, because static analysis usually encapsulates an approximate view of the program, its analysis output (bugs) has to be manually validated. The analysis logic of a static analyzer may be catered to different use cases, such as finding insecure API usages, erroneous code patterns etc. This analysis logic is usually encoded as a set of rules (*checking* rules), while the analysis itself is carried out by a static analyzer's core engine.

Static program analysis includes, among other types of analyses, program data-flow and control-flow analyses [1]. Data-flow analysis inspects the flow of data between program variables; likewise control-flow analysis inspects the flow of control in the program. While data-flow analysis may be used to understand how program input interacts with program variables, control-flow analysis may be used to understand how control is transferred from one program routine to another. In practice, both data and control flow analyses are essential components of a static analyzer.

Program data and control-flow may be analyzed at different program abstractions. In our work, we focus on syntactic as well as semantic analysis, using the program abstract syntax tree (AST), and control flow graph (CFG) respectively. At the syntactic level, our analysis is performed on the program's AST, and at the semantic level, on the program's CFG. A program's AST representation comprises syntactic elements of a program, such as the `If, For, While` statements, program variables and their data types etc. Each syntactic element is represented as an AST node. All AST nodes, with the exception of the root and the leaf nodes, are connected by edges that denote a parent-child relationship. The CFG of a program unit represents its semantic elements, such as the control flow between blocks of program statements. The CFG nodes are basic blocks: Group of program statements without a branching instruction. The CFG edges connect basic blocks that comprise a possible program path. The infrastructure to obtain program AST, CFG, and perform analysis on them is available in modern compiler toolchains.

Fuzz Testing Fuzzing is one of the most common dynamic analysis techniques used in security assessments. It was introduced by Miller et al. to evaluate the robustness of UNIX utilities [22]. Ever since, fuzzing has seen widespread adoption owing to its effectiveness in eliciting faulty program behavior. The first fuzzer functioned without any program knowledge: It simply fed random inputs to the

program. In other words, it was a blackbox (program agnostic) fuzzer. Blackbox fuzzers paved the way for modern fuzzers that are program aware.

State-of-the-art fuzzers build a model of the analyzed program as it is tested. This model is used to guide testing more optimally, i.e., expend resources for teasing out unexplored program paths. Techniques used to build a model of the program under test may vary from coverage tracing (afl) [34], to constraint solving (SAGE) [14]. Fuzzers may also expect the user to define a grammar underlying the message format being tested. Examples of such fuzzers are the Peach Fuzzer [29] and Sulley [28], both of which generate inputs based on a user specified grammar. Fuzzers such as afl support the use of message constructs for fuzzer guidance. However, unlike Peach, afl does not require a formal grammar specification; it simply uses pre-defined constructs in the input *dictionary* toward input mutation.

3 Program Analysis Guided Fuzzing

In this section, we first briefly outline our specific problem scope with regard to protocol specification inference, then provide an overview of our approach, and finally describe our methodology.

Problem Scope An application protocol specification usually comprises a *state machine* that defines valid sequences of protocol messages, and a *message format* that defines the protocol message. In our work, we focus on inferring the protocol message format only, leaving the inference of the state machine for future work. Since file formats are stateless specifications, our work is applicable for conducting security evaluations of file format parsers as well.

Approach Overview We demonstrate how fuzz testing of network applications can be significantly improved by leveraging static analysis for test guidance. It has already been suggested in non-academic circles that a carefully constructed dictionary of parser input can dramatically improve a fuzzer's effectiveness [35]. However, creating input dictionaries still requires domain expertise. We automatically generate input dictionaries by performing static program analysis, supplying it to an off-the-shelf fuzzer toward input generation. Indeed, our prototype builds on legacy fuzzers to demonstrate the effectiveness of our approach.

Figure 1 illustrates our analysis and test workflow. First, we statically analyze application source code and obtain a dictionary of protocol message constructs and conjunctions. Each item in the dictionary is an independent message fragment: It is either a simple message construct, or a conjunction of multiple constructs. For example, a constant string `SIP/2.0` in the source code is inferred as a message *construct*, while usages of another construct, say the constant string `INVITE`, that are contingent on `SIP/2.0` are inferred to be a *conjunction* of the form `INVITE SIP/2.0`. Second, we supply the input dictionary obtained in the first step to a fuzzer toward input generation. The fuzzer uses the supplied dictionary together with an initial set of program inputs (seeds) toward fuzzing an application test case. In contrast to prior work, our analysis is automatic, and requires neither a hand-written grammar specification, nor manual software al-

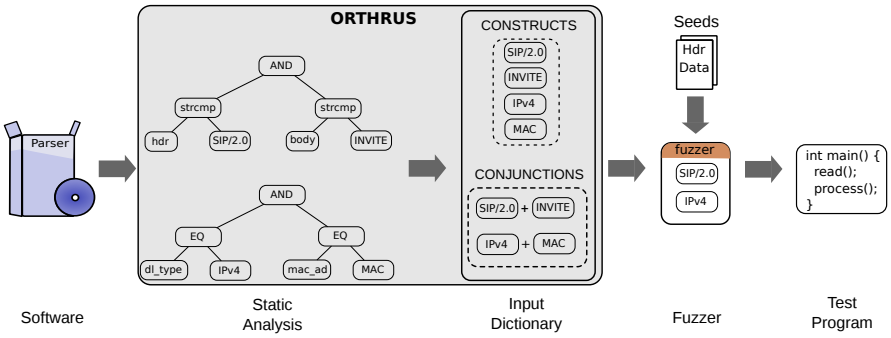


Fig. 1: Work-flow for program analysis guided fuzzing.

terations. Furthermore, the input dictionary obtained through our analysis may be supplied *as is* to existing fuzzers such as afl, aflfast, and libFuzzer, making our approach legacy compliant.

3.1 Input Dictionary Generation

The use of static program analysis for inferring program properties is a long-standing field of research. However, the main challenge underlying our approach is that our analysis must infer properties of the *program input* from application source code. Although Rice's theorem [17] states that all semantic program properties are undecidable in general, we aim to make an informed judgement.

Program Slicing The first problem we encounter is an instance of the classical forward slicing problem [12]: determining the subset of program statements, or variables that process, or contain program input. Although existing forward slicing techniques obtain precise inter-procedural slices of small programs, they do not scale up to complex network parsers that exhibit a high degree of control as well as data-flow diversity.

As a remedy, we obtain a backward program slice with respect to a predetermined set of program statements that are deemed to process program input. These program statements are called taint sinks, since program input (taint) flows into them. Since our analysis is localized to a set of taint sinks, it is tractable and scales up to large programs. Naturally, the selection criteria for taint sinks influence analysis precision, and ultimately decide the quality of inferred input fragments. Therefore, we employ useful heuristics and follow reasonable design guidelines so that taint sink selection is not only well-informed by default, but can also benefit from domain expertise when required. We explain our heuristics and design guidelines for taint sink selection in the next paragraph.

Taint Sinks We select a program statement as a taint sink if it satisfies one or more of the following conditions:

1. It is a potentially data-dependent control flow instruction, such as *switch*, `if` statements.

2. It is a well-known data sink API (e.g., `strcmp`), or an API that accepts `const` qualified arguments as input.
3. It contains a constant assignment that contains a literal character, string, or integer on the right hand side, such as
   `const char *sip = ‘‘SIP/2.0’’`

Although these heuristics are simple, they are effective, and have two useful properties that are crucial to generating an effective fuzzer dictionary. First, they capture a *handful* of potential input fragments of high relevance by focusing on program data and control flow. In contrast, a naïve textual search for string literals in the program will inevitably mix-up interesting and uninteresting use of data, e.g., strings used in print statements will also be returned. Second, although our heuristics are straightforward, they capture a wide array of code patterns that are commonly found in parsing applications. Thus, they constitute a good *default specification* that is applicable to a large class of parsing applications. The defaults that are built-in to our analysis framework make our solution accessible for conducting security assessments of third-party network software.

Naturally, our heuristics may miss application-specific taint sinks. A prominent example is the use of application specific APIs for input processing. As a remedy, we permit the security analyst to specify additional taint sinks as an analysis parameter. In summary, we facilitate entirely automatic analysis of third-party software using a default taint specification, while opportunistically benefiting from application-specific knowledge where possible. This makes our analysis framework flexible in practice.

Analysis Queries In order to infer protocol message constructs, we need to analyze data and control-flow around taint sinks. To facilitate fast and scalable analysis, we design a query system that is capable of both syntactic and semantic analysis. Fortunately, the infrastructure to obtain program AST, CFG, and perform analysis on them is already available in modern compiler toolchains. Thus, we focus on developing the analysis logic for performing backward program slicing toward obtaining protocol message constructs.

Algorithm 1 illustrates our analysis procedure for generating an input dictionary from source code. We begin by initializing our internal data-structures to an empty set (lines $2-4$). Next, we iterate over all compilable source files in the code repository, and obtain their program AST and CFG representations (lines $8-9$) using existing compiler routines. Based on our default set of taint sinks, we formulate syntactic and semantic queries (described next) that are designed to elicit input message constructs or their conjunctions in source code (line 6). Using these queries, we obtain a set of input message constructs using syntactic analysis (line 11), and a set of input message conjunctions using semantic analysis (line 13) in each source file. The constructs and conjunctions so obtained are added to the dictionary data structure (line $14-15$) and the analysis continues on the next source file.

Syntactic Queries At the syntactic level, our analysis logic accepts functional queries and returns input message constructs (if any) that match the issued

Algorithm 1 Pseudocode for generating an input dictionary.

```
1: function GENERATE-DICTIONARY(SourceCode, Builder)
2:     dictionary = ∅
3:     constructs = ∅
4:     conjunctions = ∅
5:     ▷ Queries generated from internal database
6:     queries = Q
7:     for each sourcefile in SourceCode do
8:         ast = frontendParse(sourcefile)
9:         cfg = semanticParse(ast)
10:        ▷ Obtain constructs
11:        constructs = syntactic-analysis(ast, queries)
12:        ▷ Obtain conjunctions of existing constructs
13:        conjunctions = semantic-analysis(cfg, constructs)
14:        ▷ Update dictionary
15:        dictionary += constructs
16:        dictionary += conjunctions
17:    return dictionary
18:
19: function SYNTACTIC-ANALYSIS(AST, Queries)
20:    constructs = ∅
21:    for each query in Q do
22:        constructs += synQuery(AST, query)
23:    return constructs
24:
25: function SYNQUERY(AST, Query)
26:    matches = ∅
27:    while T = traverseAST(AST) do
28:        if Query matches T then
29:            matches += (T.id, T.value)
30:    return matches
31:
32: function SEMANTIC-ANALYSIS(CFG, Constructs)
33:    conjunctions = ∅
34:    ▷ Obtain conjunctions in a given calling context
35:    conjunctions += Context-Sensitive-Analysis(CFG, Constructs)
36:    ▷ Obtain productions in a given program path
37:    conjunctions += Path-Sensitive-Analysis(CFG, Constructs)
38:    return conjunctions
```

query. These queries are made against the program AST. A functional query is composed of boolean predicates on a program statement or data type. As an example, consider the following query:

`stringLiteral(hasParent(callExpr(hasName(‘‘strcmp’’)))).`

The query shown above searches for a program value of type string (`stringLiteral`) whose parent node in the AST is a function call (`callExpr`), and whose declaration name is `strcmp`. Thus, a functional query is essentially compositional in nature and operates on properties of the program AST. There are two key benefits of functional queries. First, their processing time is very low allowing them to scale up to large codebases (see §4.1). Second, since large parsing applications use a recurring pattern of code to parse input messages of different formats, even simple queries can be efficient at building a multi-protocol input dictionary.

Syntactic queries are useful for obtaining a list of simple input message constructs such as constant protocol keywords. However, these queries do not analyze the context in which constructs appear in the program. Analyzing the

context brings us a deeper understanding of the input message format. As an example, we may know which two constructs are used in conjunction with each other, or if there is a partial order between grammar production rules involving these constructs. Deeper analysis of message constructs may infer complex message fragments, allowing the fuzzer to explore intricate parsing routines. To facilitate such context-sensitive analyses, we write context and path-sensitive checkers that enable semantic queries.

Semantic Queries At the semantic level, a query accepts a list of input message constructs as input, and returns conjunctions (if any) of constructs as output. Semantic queries are made against a context-sensitive inter-procedural graph [30] constructed on a program's CFG. Each query is written as a checker routine that returns the set of conjunctions that can be validated in the calling context where the input construct appeared. As an example, consider the parsing code snippet shown in Listing 1.1.

Listing 1.1: Sample parser code.

```
1 int parse(const char *token1, const char *token2) {
2   if (token1 == "INVITE")
3     if (strcmp(token2, "SIP/2.0"))
4       do_something();
5 }
```

The `parse` function takes two string tokens as input and performs an operation only when the first token is `INVITE` and the second token is `SIP/2.0`. From this code, we can infer that there is a dependency between the two tokens, namely, that `INVITE` is potentially followed by the `SIP/2.0` string. While syntactic queries can only identify simple message constructs, semantic queries can be used to make an inference about such message conjunctions. Together, syntactic and semantic queries may be used to build a dictionary of the input message format.

Implementation We have implemented our approach in a research prototype, that we call Orthrus. Our query system is composed of tooling based on the libASTMatchers, and the libTooling infrastructure in Clang (syntactic queries), and checkers to the Clang Static Analyzer [20] (semantic queries).

3.2 Dictionary Based Fuzzing

An input dictionary can improve the effectiveness of fuzzing by augmenting the program representation maintained by the fuzzer for test guidance. The input fragments in the supplied dictionary enable input mutations that are well-informed, and in some cases more effective at discovering new program paths than purely random mutations. Contemporary fuzzers offer an interface to plug in an application-specific dictionary. We use this interface to supply the input fragments inferred by our analysis framework to the fuzzer.

Algorithm 2 presents the pseudocode for dictionary based fuzzing employed by most present-day fuzzers. Dictionary based mutations may be performed either deterministically (at all byte offsets in the input stream, line $4-5$), or

Algorithm 2 Pseudocode for dictionary-based fuzzing.

```
1: function DICTIONARY-FUZZ(input, Dictionary, deterministic)
2:     dictToken = Random(Dictionary)
3:     if deterministic then
4:         for each byteoffset in input do
5:             fuzz-token-offset(input, dictToken, byteoffset)
6:     else
7:         byteoffset = Random(sizeOf(input))
8:         fuzz-token-offset(input, dictToken, byteoffset)
9:
10: function FUZZ-TOKEN-OFFSET(input, dictToken, byteoffset)
11:     ▷ Token overwrites input byte
12:     input[byteoffset] = dictToken
13:     Program(input)
14:     ▷ Token inserted into input
15:     InsertToken(input, byteoffset, dictToken)
16:     Program(input)
```

non-deterministically (at a random byte offset, line $7-8$). There are two kinds of dictionary based mutations used by fuzzers: overwrite, and insert. In an overwrite operation, the chosen dictionary token is used to overwrite a portion of a program input in the fuzzer queue (line $12-13$). In an insert operation, the chosen token is inserted into the queued input at the specified offset (line $15-16$). Typically, fuzzers perform both mutations on a chosen token.

Fuzzers bound the runtime allocated to dictionary-based fuzzing routines. In practice, fuzzers either use up to a certain threshold (typically a few hundred) of supplied dictionary tokens deterministically, while using the rest probabilistically, or pick each token at random. Thus, it is important that the size of the supplied dictionary is small, and the relevance of the tokens is high. Our use of demand-driven queries, and analyses of varying precision ensures that we supply such a dictionary to the fuzzer.

4 Evaluation

In this section, we present our evaluation of Orthrus in both controlled and uncontrolled environments. First, we (i) Quantitatively evaluate our analysis run time towards dictionary generation, and (ii) Qualitatively evaluate the generated dictionary tokens, for the codebases under test (§4.1). Second, we measure the time to uncover vulnerabilities using Orthrus generated dictionaries in a set of fuzzer benchmarks (§4.2). Third, we measure the test coverage achieved and examine the vulnerabilities exposed by fuzzing production code with the aid of Orthrus generated dictionaries (§4.3). We conclude this section with a discussion of factors that may limit the validity of our approach and how we address them.

Measurement Infrastructure All measurements presented in this section were performed on a 64-bit machine with 80 CPU threads (Intel Xeon E7-4870) clocked at 2.4 GHz, and 512 GB RAM.

Software	Source Lines of Code	Compilation	Dictionary Generation		
			Syntactic	Semantic	Total
c-ares	97k	2.11s	0.43s	20.14s	20.57s
libxml2	196k	17.95s	1.48s	23.09s	24.57s
openssl	278k	20.02s	6.45s	5m 37.24s	5m 43.69s
nDPI	27k	7.16s	2.14s	42.84s	44.98s
tcpdump	75k	2.99s	0.32s	9.04s	9.36s
woff2	39k	3.20s	3.58s	11.58s	15.16s

Table 1: Dictionary generation run time relative to code compilation time. Timing measurements have been averaged over ten runs and are presented in minutes(m) and seconds(s).

Software	Taint Sink	Query	Input Fragments
libxml2	`xmlBufferWriteChar()`, `xmlOutputBufferWrite()`	Obtain constant argument	`xml:lang="`, `<!DOCTYPE`, `<![CDATA[`, `xmlns`
nDPI	`memcmp()`, `strcmp()`	Obtain constant argument	`snort`, `America Online Inc.`, `last message`

Table 2: A sample of string input fragments extracted from the source code of libxml2, and nDPI using syntactic queries. Extracted fragments are comma separated.

4.1 Analysis Run Time and Effectiveness

Table 1 presents the run times of static analysis (both syntactic and semantic) performed for dictionary generation for each of the code bases evaluated in this paper. To put the run times in perspective, the run time of code compilation for each code base is presented in the third column. Since semantic analysis is computationally more expensive than syntactic analysis, it dominates the dictionary generation run time . However, in relation to fuzzing run time that is usually in the order of days, the time required for dictionary generation (at most a few minutes across our data-set) is negligible.

Table 2 presents a sample of input fragments (constructs) extracted from the source code for libxml2, and nDPI for which dictionary-based fuzzing showed substantial improvement in test coverage and outcome. In the interest of space and visual clarity, we have excluded fragments extracted from tcpdump since they mainly comprise binary input. Listing 1.2 shows one of the syntactic queries applied to the nDPI, and libxml2 codebases that resulted in the sample fragments presented in Table 2. Our analysis heuristics have helped build an XML input dictionary that is similar in content to the manually created XML dictionary for

afl. Moreover, using backward slicing from familiar taint sinks such as `memcmp`, we have been able to extract protocol fragments (such as the string literal `America Online Inc.` used by nDPI to fingerprint instant messaging traffic) that have been instrumental in increasing test coverage.

Listing 1.2: Syntactic query issued on nDPI and libxml2 codebases. The query returns string literals passed as arguments to taint sinks such as `strcmp`.

```
// Obtain string literals passed to POSIX APIs "strcmp", and
// "memcmp", and libxml2 APIs "xmlBufferWriteChar", and
// "xmlOutputBufferWrite".
StatementMatcher StringMatcher =
  stringLiteral(
                hasAncestor(
                            declRefExpr(
                                to(namedDecl(
                                 anyOf(hasName("strcmp"),
                                       hasName("memcmp"),
                                       hasName("xmlBufferWriteChar"),
                                       hasName("xmlOutputBufferWrite")
                                      )
                                  )
                                )
                            )
                )
  ).bind("construct");
```

4.2 Benchmarks: Time to Vulnerability Exposure

To enable independent reproduction, we briefly document our evaluation methodology.

Fuzzer Test Suite In order to measure the time required to expose program vulnerabilities, we used the fuzzer test suite [15]. The fuzzer test suite is well-suited for this purpose because it provides a controlled environment in which timing measurements can be done, and contains test cases for several known high-profile vulnerabilities. Indeed, the test suite has been used for benchmarking the LLVM libFuzzer [21], that we use as a baseline in our evaluation. The specific vulnerabilities in the test suite that feature in our evaluation are: CVE-2014-0160 [23] (OpenSSL Heartbleed), CVE-2016-5180 [25] (buffer overflow in the c-ares dns library), CVE-2015-8317 [24] (buffer overflow in libxml2), and a security-critical bug in Google's WoFF2 font parser [6].

Test Methodology For each test case, our evaluation was performed by measuring the time to expose the underlying vulnerability in two scenarios: (i) The baseline fuzzer alone; and (ii) The baseline fuzzer augmented with an Orthrus generated dictionary. Our approach is deemed effective when the time to expose vulnerability reduces in comparison to the baseline, and is ineffective/irrelevant when it increases or remains the same in comparison to the baseline. Timing measurements were done using Unix's `time` utility. In order to reduce the effect of seemingly random vulnerability exposures, we obtained at least 80 timing measurements for each test case in both scenarios. Measurements for each test

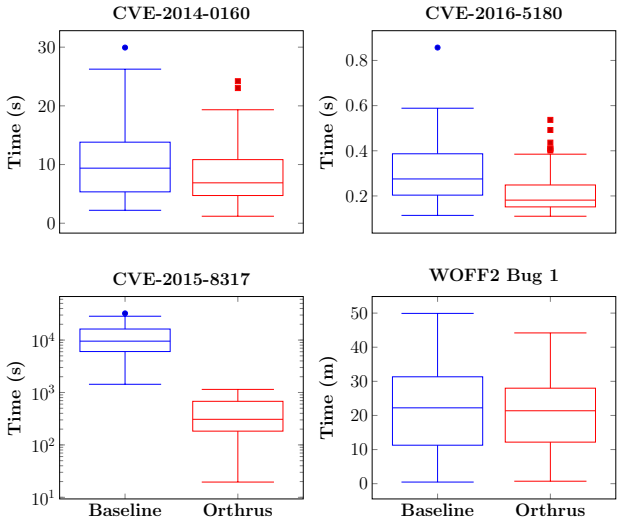


Fig. 2: Comparison of time required to expose vulnerability using libFuzzer as the baseline.

case were carried out in parallel, with each experiment being run exclusively on a single core. The input dictionary generated by Orthrus was supplied to libFuzzer via the `-dict` command line argument. Finally, to eliminate the effect of seed corpuses on measurement outcome, we strictly adhered to the selection of seed corpuses as mandated by the fuzzer test suite documentation.

Results Figure 2 presents our test results as box plots. The baseline box plot (libFuzzer) is always on the left of the plot, and results for libFuzzer augmented with Orthrus (Orthrus) on the right. The Orthrus generated input dictionary brought down the time to expose a buffer overflow in the libxml2 library (CVE-2015-8317) by an order of magnitude (from a median value of close to 3h using the baseline to a median value of 5 minutes using our approach). For all the other test cases, the median time to expose vulnerability was lower for Orthrus in comparison to libFuzzer. In addition, Orthrus shrunk the range of timing variations in exposing the vulnerability.

To understand the varying impact of the supplied dictionary on the time to vulnerability exposure, we studied each of the tested vulnerabilities to understand their root cause. Our approach consistently brought down the time to exposure for all vulnerabilities that were triggered by a file or protocol message specific to the application under test. Thus, our approach worked well in scenarios where knowledge of the input format was crucial to eliciting the vulnerability. Furthermore, in scenarios where our approach did not substantially lower the time to vulnerability exposure, the time penalty incurred by our approach, owing to the test time dedicated to dictionary mutations, was marginal.

In summary, we find that static program analysis can improve bug-finding *efficiency* of fuzzers for those class of bugs that are triggered by highly structured input (commonly found in network applications, and file format parsers), while not imposing a noticeable performance penalty.

4.3 Case Study

To investigate the practical utility of Orthrus, we conducted a case study of two popular network applications, namely, nDPI, and tcpdump. These applications were selected because they are not only deployed in security-critical environments but also parse potentially attacker-controlled data. For each application, we conducted multivariate testing using baseline fuzzers such as afl and aflfast [3] with and without an Orthrus generated dictionary.

The chosen applications were also fuzzed using the Peach fuzzer [29], a state-of-the-art fuzzer for protocol security assessments. Since grammar specifications for the set of protocols parsed by tcpdump, and nDPI were not publicly available, we enabled Peach fuzzer's input analyzer mode that automatically infers the input data model. Such an evaluation was aimed at comparing Peach fuzzer with Orthrus in scenarios where a data model specification is not available. However, the community edition of the Peach fuzzer that we had access to, is not geared toward long runs. In our Peach-based experiments, we could not achieve a run time of longer than 24 hours. This prevents a fair comparison of the two approaches. Therefore, we document results of our Peach experiments for reference, and not a comparative evaluation.

Evaluation Methodology We evaluated Orthrus using two metrics, namely, test coverage achieved, and the number of program vulnerabilities exposed. Test coverage was measured as the percentage of program branches that were discovered during testing. Since fuzzers often expose identical crashes, making it non-trivial to document unique vulnerabilities, we semi-automatically deduplicated fuzzer crashes in a two-step process. First, we used the concept of fuzzy stack hashes [26] to fingerprint a crash's stack trace using a cryptographic hash function. Second, crashes with a unique hash were manually triaged to determine the number of unique program vulnerabilities. We used two elementary seeds (bare-bone IPv4, and IPv6 packets) to fuzz tcpdump, and nDPI. Tests involving the fuzzers afl and aflfast were conducted in a multi-core setting.

Fuzzing Duration Dictionary based mutations get a fraction of the total fuzz time of a fuzzer. Thus, to fully evaluate our approach, we ran the fuzzer configurations (except Peach) until each unique program input synthesized by the fuzzer was mutated with the supplied dictionary constructs at least once. Owing to the relatively poor execution throughput of the evaluated software (under 100 executions per second), we had to run each fuzzer over a period of 1 week in which time the supplied dictionary was utilized at least once for each unique input.

Table 3: Test coverage achieved (in %) by different fuzzing configurations.

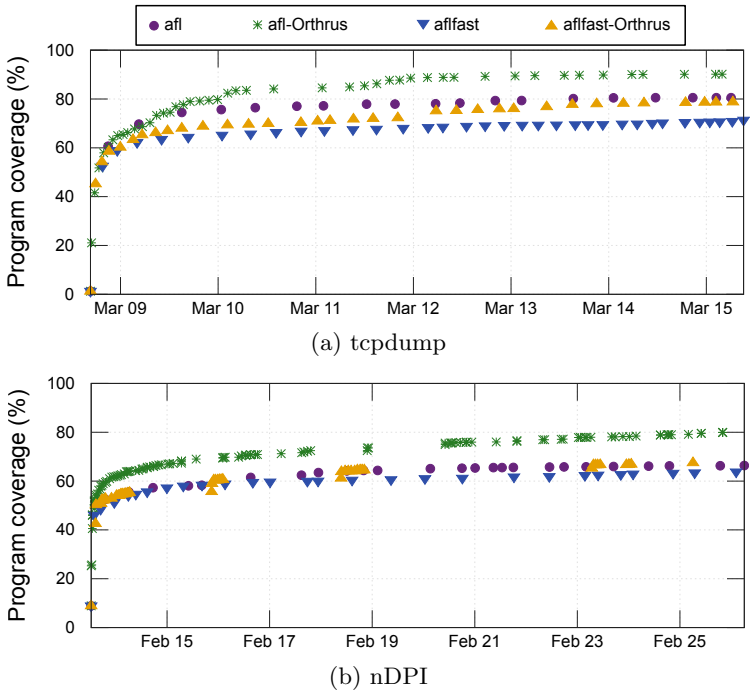
Software	afl	afl-orthrus	aflfast	aflfast-orthrus	Peach-analyzer
tcpdump	80.56	90.23 (+ 9.67)	71.35	78.82 (+7.47)	6.25
nDPI	66.92	81.49 (+14.57)	64.40	68.10 (+3.70)	24.98

Utilities CERT's `exploitable` [11] utility was used for crash deduplication. We used AddressSanitizer [2] as a debugging aid; this expedited the bug reporting process.

Evaluated Software We evaluated nDPI revision f51fef6 (November 2016), and tcpdump trunk (March 2017).

Test Coverage Our test coverage measurements present the fraction of all program branches (edges) covered by test cases generated by a fuzzer configuration. We have evaluated Orthrus against two baselines, namely, afl, and aflfast. Therefore, our measurements have been obtained for afl, afl augmented with Orthrus-generated input dictionary (afl-Orthrus), aflfast, aflfast augmented with Orthrus-generated input dictionary (aflfast-Orthrus), and the Peach fuzzer with a binary analyzer data model. Table 3 shows the test coverage achieved by different fuzzer combinations for tcpdump, and nDPI, while Figure 3 visualizes code coverage over time. Program coverage was measured when there was a change in its magnitude. Due to the relatively short running duration of the Peach fuzzer, we have excluded its coverage visualization.

As shown in Figure 3, the obtained coverage measurements for tcpdump, and nDPI, approach a saturation point asymptotically. For both tcpdump, and nDPI, the growth rate in test coverage is higher initially, tapering off asymptotically to zero. The test coverage curves for afl-Orthrus and aflfast-Orthrus have a higher initial growth rate compared to their respective baselines, namely, afl, and aflfast. This results in a consistent increase in overall test coverage achieved by Orthrus in comparison to the baseline fuzzers, as shown in Table 3. For nDPI, Orthrus' input dictionary increases test coverage by 14.57% over the afl fuzzer. In the case of tcpdump, this increase in test coverage is 9.67%. Orthrus' enhancements in test coverage over aflfast for nDPI, and tcpdump are 3.7%, and 7.47% respectively. Although aflfast is a fork of afl, the supplied input dictionary has a lesser effect on the former than the latter. To understand this anomaly, we examined the source code of afl, and aflfast. afl performs dictionary-based mutations on *all* inputs in the fuzzer queue at least once. However, aflfast performs dictionary-based mutations on a given input in the queue, only when the input's *performance score* (computed by the aflfast algorithm) is above a certain threshold. We determined that the threshold used by aflfast is too aggressive, resulting in too few inputs in the fuzzer queue undergoing dictionary mutations.



(a) tcpdump

(b) nDPI

Fig. 3: Test coverage as a function of time for tcpdump 3a, and nDPI 3b, for different fuzzing configurations. Program coverage measurements were made only when there was a change in its magnitude.

Vulnerabilities Exposed Table 4 shows the number of vulnerabilities exposed in nDPI, and tcpdump, across all fuzzing configurations. In the case of tcpdump, the positive impact of the Orthrus generated dictionary is evident. afl, and afl-Orthrus, exposed 15, and 26 unique vulnerabilities respectively. 10 out of the 11 additional vulnerabilities exposed by afl-Orthrus, were exclusively found by it, i.e., it exposed 10 vulnerabilities in tcpdump not found by stand-alone afl. aflfast, and aflfast-Orthrus configurations exposed 1 and 5 vulnerabilities respectively. aflfast-Orthrus exposed 4 vulnerabilities that were not exposed by stand-alone aflfast. In the case of nDPI, afl-Orthrus exposed 4 vulnerabilities that were not found by stand-alone afl, while aflfast-Orthrus exposed 1 such vulnerability. For both nDPI, and tcpdump, aflfast-Orthrus finds fewer number of vulnerabilities overall in comparison to its baseline. We conjecture that the fuzz schedule alterations carried out in aflfast [3] influence the scheduling of dictionary-mutations, resulting in the observed drop.

Table 5 documents those vulnerabilities found using Orthrus generated dictionaries that were not found by stand-alone fuzzing of tcpdump, and nDPI. The number of exposed vulnerabilities that may be exclusively attributed to Orthrus are 10, and 5, for tcpdump, and nDPI respectively. Overall, Orthrus

Table 4: Number of bugs and vulnerabilities exposed by different fuzzing configurations. For Orthrus-based fuzzer configurations, the number of bugs exclusively found by them is shown in brackets.

Software	afl	afl-orthrus	aflfast	aflfast-orthrus	Peach-analyzer
tcpdump	15	26 (+10)	1	5 (+ 4)	0
nDPI	26	27 (+ 4)	24	17 (+ 1)	0

Table 5: Vulnerabilities exposed exclusively using Orthrus generated dictionaries in afl, and aflfast, for tcpdump, and nDPI. All the vulnerabilities result in a buffer overflow. Number in square brackets indicates the number of vulnerabilities found.

Software	Vulnerable Component
tcpdump	IPv6 DHCP packet printer
	IPv6 Open Shortest Path First (OSPFv3) packet printer
	IEEE 802.1ab Link Layer Discovery Protocol (LLDP) packet printer
	ISO CLNS, ESIS, and ISIS packet printers [**2**]
	IP packet printer
	ISA and Key Management Protocol (ISAKMP) printer
	IPv6 Internet Control Message Protocol (ICMPv6) printer
	Point to Point Protocol (PPP) printer
	White Board Protocol printer
nDPI	ZeroMQ Message Transport Protocol processor
	Viber protocol processor
	Syslog protocol processor
	Ubiquity UBNT AirControl 2 protocol processor
	HTTP protocol processor

generated dictionaries exposed vulnerabilities in 14 different network protocols across the two codebases. Some of the exposed vulnerabilities are in the processing of proprietary protocol messages such as the Viber protocol. All the exposed vulnerabilities resulted in buffer overflows, and were immediately reported to the respective vendors. These results are a testament to the efficacy of our approach in increasing the breadth of testing for complex network applications without requiring domain-specific knowledge.

Preliminary Results for Snort++ We used Orthrus to perform dictionary-based fuzzing of `snort++`, a `C++` implementation of the popular snort IDS. Baseline fuzzing with afl-fuzz helped find a single vulnerability (CVE-2017-6658) in the `snort++` decoder implementation. In contrast, the Orthrus generated dictionary has helped find an additional vulnerability (CVE-2017-6657) in the LLC packet decoder implementation of `snort++` [31].

4.4 Limitations

Although our evaluations show that static analysis guided fuzzing is beneficial, our positive results may not generalize to other parsing applications. However, our evaluation comprising six different parser implementations provides strong evidence that our approach can make fuzz testing more effective. Automatically generated parsers (e.g., yacc-based parsers) may contain code that is structurally different than hand-written parsers that we have evaluated. We believe that their analysis may be carried out at the specification level than at the source code level. Furthermore, we make use of simple heuristics to infer input message fragments from source code. Thus, our analysis may miss legitimate input fragments (false negatives), and/or add irrelevant tokens to the input dictionary (false positives). However, we take practical measures to keep the number of false positives/negatives low. For example, our design incorporates practical security advice given by reputed institutes such as CERT [5] that have been compiled over years of source code audits. In our case study, we make use of a small (yet relevant) seed set to bootstrap fuzzing. It is possible that a diverse seed set improves the performance of our baseline fuzzers. Having said that, we have carefully analyzed the additional coverage achieved solely through the use of the supplied dictionary to ensure that the presented increments can be attributed to our method. In addition, we have manually triaged all vulnerabilities found exclusively using dictionary-based fuzzing to ensure causality, i.e., they were ultimately exposed due to the use of specific tokens in the supplied dictionary.

5 Related Work

Multiple techniques have been proposed to improve the effectiveness of fuzzing. For our discussion of related work, we focus on approaches that infer the protocol specification, use grammar-based fuzzing, or query-driven static analysis approaches.

Inferring Protocol Specification There are two problems underlying protocol specification inference: Inferring the protocol (i) Message format; and (ii) State machine. Prior work, with the exception of Prospex [7] has focused solely on the message format inference problem. Broadly, two approaches have been proposed to automatically infer the protocol specification. The first approach relies entirely on network traces for performing the inference, exemplified by the tool Discoverer [8]. As other researchers have noted, the main problem with this approach is that network traces contain little semantic information, such as the relation between fields in a message. Therefore, inference based entirely on network traces is often limited to a simple description of the message format that is an under-approximation of the original specification. The second approach, also a pre-dominant one, is to employ dynamic program analysis in a setting where the network application processes sample messages, in order to infer the protocol specification. Proposals such as Polyglot [4], Tupni [9], Autoformat [19], Prospex [7], and the tool by Wondracek et al. [32] fall into this category. In comparison to our work, these proposals have two shortcomings. First, they require

dynamic instrumentation systems that are often proprietary or simply inaccessible. Dynamic instrumentation and analysis often requires software expertise, making it challenging for auditing third-party code. In contrast, we show that our analysis can be bundled into an existing compiler toolchain so that performing protocol inference is as simple as compiling the underlying source code. Second, prior work with the exception of Prospex, have not specifically evaluated the impact of their inference on the effectiveness of fuzz testing. Although Comparetti et al. [7] evaluate their tool Prospex in conjunction with the Peach fuzzer, their evaluation is limited to finding known vulnerabilities in controlled scenarios. In contrast to these studies, we extensively evaluate the impact our inference on the effectiveness of fuzzing, both quantitatively in terms of test coverage achieved, and time to vulnerability exposure, and qualitatively in terms of an analysis of vulnerabilities exclusively exposed using our inference in real-world code.

Grammar-based Fuzzing Godefroid et al. [13] design a software testing tool in which symbolic execution is applied to generate grammar-aware test inputs. The authors evaluate their tool against the IE7 JavaScript interpreter and find that grammar-based testing increases test coverage from 53% to 81%. Although their techniques are promising, their work suffers from three practical difficulties. First, a manual grammar specification is required for their technique to be applied. Second, the infrastructure to perform symbolic execution at their scale is not publicly available, rendering their techniques inapplicable to third-party code. Third, their approach requires non-trivial code annotations, requiring a close co-operation between testers and developers, something that might not always be feasible. In contrast, we solve these challenges by automatically inferring input data formats from the source code. Indeed, we show that more lightweight analysis techniques can substantially benefit modern fuzzers. Langfuzz [16] uses a grammar specification of the JavaScript and PHP languages to effectively conduct security assessments on the respective interpreters. Like Godefroid et al., the authors of Langfuzz demonstrate that, in scenarios where a grammar specification can be obtained, specification based fuzzing is superior to random testing. However, creating such grammar specifications for complex network applications manually is a daunting task. Indeed, network protocol specifications (unlike computing languages) are specified only semi-formally, requiring protocol implementors to hand-write parsers instead of generating them from a parser generator. Such practical difficulties make grammar (specification) based fuzzing challenging for network applications.

Query Based Program Analysis Our static analysis approach is inspired by prior work on the use of queries to conduct specific program analyses by Lam et al. [18], and automatic inference of search patterns for discovering taint-style vulnerabilities from source code by Yamaguchi et al. [33]. At their core, both these works use a notion of program queries to elicit vulnerable code patterns from source code. While Lam et al. leverage datalog queries for analysis, Yamaguchi et al. employ so called *graph traversals*. In contrast to their work, we leverage query-driven analysis toward supporting a fuzzer instead of attempting static vulnerability discovery.

6 Conclusions and Future Work

In this paper, we demonstrate how static analysis guided fuzzing can improve the effectiveness of modern off-the-shelf fuzzers, especially for networking applications. Code patterns indicate how user input is processed by the program. We leverage this insight for gathering input fragments directly from source code. To this end, we couple a static analyzer to a fuzzer via an existing interface. Using input dictionaries derived from semantic and syntactic program analysis queries, we are able to not only increase the test coverage of applications by 10–15%, but also reduce the time needed to expose vulnerabilities by an order of magnitude in comparison to fuzzers not supplied with an input dictionary. We leverage our research prototype to fuzz two high-profile network applications, namely, nDPI, a deep packet inspection library, and tcpdump, a network packet analyzer. We find 10 zero-day vulnerabilities in tcpdump, and 5 zero-day vulnerabilities in nDPI that were missed by stand-alone fuzzers. These results show that our approach holds promise for making security assessments more effective.

Our work highlights the need for a stronger interaction between program analysis and testing. Although our study describes one way in which program analysis can enhance fuzzing, exploiting their reciprocal nature poses some interesting problems such as directing static analysis on code portions that have not been fuzzed. This is one avenue for future work. A logical follow up of our work will be to infer the protocol state machine in addition to its message format, and leverage the additional insight for conducting stateful fuzzing. Leveraging our inference algorithm toward conducting large-scale analysis of open-source C/C++ parser implementations is another avenue for future work that will shed light on the security dimension of an important software component. Indeed, targeting our analysis at the binary level will help us evaluate its efficacy against closed source applications.

Acknowledgements. We would like to thank Julian Fietkau for helping customize the Peach fuzzer for our experiments. This work was supported by the following awards and grants: Bundesministerium für Bildung und Forschung (BMBF) under Award No. KIS1DSD032 (Project Enzevalos), Leibniz Prize project by the German Research Foundation (DFG) under Award No. FKZ FE 570/4-1, the Helmholtz Research School in Security Technologies scholarship, and the Danish Villum project ReNet. The opinions, views, and conclusions contained herein are those of the author(s) and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of BMBF, DFG, or, any other funding body involved.

Bibliography

[1] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques. Addison wesley Boston (1986)

[2] Address Sanitizer. https://clang.llvm.org/docs/AddressSanitizer.html, accessed: 03/27/17

[3] Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. In: Proc. ACM Conference on Computer and Communications Security (CCS). pp. 1032–1043. ACM (2016)

[4] Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In: Proc. ACM Conference on Computer and Communications Security (CCS). pp. 317–329 (2007)

[5] Cert secure coding standards. https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards, accessed: 06/01/17

[6] Clusterfuzzer: Heap-buffer-overflow in read. https://bugs.chromium.org/p/chromium/issues/detail?id=609042, accessed: 03/23/17

[7] Comparetti, P.M., Wondracek, G., Kruegel, C., Kirda, E.: Prospex: Protocol specification extraction. In: Proc. IEEE Security & Privacy. pp. 110–125 (2009)

[8] Cui, W., Kannan, J., Wang, H.J.: Discoverer: Automatic protocol reverse engineering from network traces. In: Proc. Usenix Security Symp. vol. 158 (2007)

[9] Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: Automatic reverse engineering of input formats. In: Proc. ACM Conference on Computer and Communications Security (CCS). pp. 391–402 (2008)

[10] Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proc. OSDI (2000)

[11] Foote, J.: The exploitable GDB plugin. https://github.com/jfoote/exploitable (2015), accessed: 03/23/17

[12] Gallagher, K.B., Lyle, J.R.: Using program slicing in software maintenance. IEEE Transactions on Software Engineering 17(8), 751–761 (1991)

[13] Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: ACM SIGPLAN Notices. vol. 43, pp. 206–215 (2008)

[14] Godefroid, P., Levin, M.Y., Molnar, D.: Sage: whitebox fuzzing for security testing. ACM Queue 10(1), 20 (2012)

[15] Google Inc.: Fuzzer test suite. https://github.com/google/fuzzer-test-suite, accessed: 03/23/17

[16] Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Proc. Usenix Security Symp. pp. 445–458 (2012)

[17] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition) (2006)

[18] Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Proc. ACM Symposium on Principles of Database Systems. pp. 1–12 (2005)
[19] Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic protocol format reverse engineering through context-aware monitored execution. In: Proc. Symposium on Network and Distributed System Security (NDSS). pp. 1–15 (2008)
[20] LLVM Compiler Infrastructure: Clang Static Analyzer. http://clang-analyzer.llvm.org/, accessed: 03/23/17
[21] LLVM Compiler Infrastructure: libFuzzer: a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html, accessed: 03/23/17
[22] Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Communications of the ACM 33(12), 32–44 (1990)
[23] MITRE.org: CVE-2014-0160: The Heartbleed Bug. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, accessed: 03/23/17
[24] MITRE.org: CVE-2015-8317: Libxml2: Several out of bounds reads. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8317, accessed: 03/23/17
[25] MITRE.org: CVE-2016-5180: Project c-ares security advisory. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5180, accessed: 03/23/17
[26] Molnar, D., Li, X.C., Wagner, D.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proc. Usenix Security Symp. vol. 9, pp. 67–82 (2009)
[27] nDPI: Open and Extensible LGPLv3 Deep Packet Inspection Library. http://www.ntop.org/products/deep-packet-inspection/ndpi/, accessed: 03/23/17
[28] OpenRCE: sulley . https://github.com/OpenRCE/sulley, accessed: 03/23/17
[29] Peach Fuzzer. http://www.peachfuzzer.com/, accessed: 03/23/17
[30] Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 49–61 (1995)
[31] Snort++ vulnerabilities found. http://blog.snort.org/2017/05/snort-vulnerabilities-found.html, accessed: 06/05/17
[32] Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic network protocol analysis. In: Proc. Symposium on Network and Distributed System Security (NDSS) (2008)
[33] Yamaguchi, F., Maier, A., Gascon, H., Rieck, K.: Automatic inference of search patterns for taint-style vulnerabilities. In: Proc. IEEE Security & Privacy. pp. 797–812 (2015)
[34] Zalewski, M.: american fuzzy lop. http://lcamtuf.coredump.cx/afl/, accessed: 03/23/17
[35] Zalewski, M.: afl-fuzz: making up grammar with a dictionary in hand. https://lcamtuf.blogspot.de/2015/01/afl-fuzz-making-up-grammar-with.html (2015), accessed: 03/23/17